The Thesis committee for Riley Jacob Wood certifies that this is the approved version of the following thesis:

RDDR: N-Versioning of Microservices

SUPERVISING COMMITTEE:

Mohit Tiwari, Supervisor

Mattan Erez

RDDR: N-Versioning of Microservices

by

Riley Jacob Wood

Thesis

Presented to the Faculty of the Graduate School

of the University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2020

Acknowledgments

First of all, thank you to Mohit Tiwari for taking me on as a student researcher. I've learned a lot under your guidance and am very grateful for the opportunities you've given me here at UT.

Thanks to Prateek Sahu for keeping me company in our office and joining me at the gym in the evenings. You were also very helpful when it came to debugging Kubernetes, so thank you!

Finally, a big thank you to Tony Espinoza who came up with the idea for this project and with whom I collaborated. Tony and I worked together on the writeup for this project, so some of his writing is reproduced in this thesis. His contributions appear in the Introduction and Related Work sections. Thank you Tony!

Abstract

RDDR: N-Versioning of Microservices

by

Riley Jacob Wood, M.S.E.

The University of Texas at Austin, 2020

SUPERVISOR: Mohit Tiwari

N-versioning is a well-studied method to increase the reliability of software. In this paper, we study n-versioning as applied to microservice-based applications. We construct a generic proxy called RDDR that orchestrates and monitors N variants of a microservice in order to detect bugs that make them behave differently. We showcase RDDR's ability to close five exemplary information leaks, where diversity is derived from: different software versions, different implementations of the same logical service, and variation provided by the OS like ASLR. These case studies feature information leakage through both frontend and backend interfaces of various web applications. To show that RDDR can close vulnerabilities while handling large volumes of benign traffic, we also apply RDDR to components of GitLab, a complex cloud application. Finally, we quantify the performance overhead associated with deploying RDDR. Our findings indicate that RDDR can patch information leaks while incurring approximately 3x CPU and memory overhead for a deployment with 3 redundant instances as expected, with modest impact to throughput and latency.

Table of Contents

List of Tables **1**

List of Figures **3**

1 Background **4**

1.1 Introduction 4

1.2 Motivation 7

1.2.1 The Applicability of N-versioning to OWASP's Top 10 Web App Vulnerabilities 7

1.2.2 Scenarios Where N-Versioning Can Be Beneficial to Cloud 15

1.3 Related Work 17

1.3.1 SOA 17

1.3.2 N-versioning 18

1.3.3 MTD 19

1.3.4 N-versioning and MTD in Other Domains 20

1.3.5 Currently Available Technology 21

2 Our System **22**

2.1 Design 22

2.1.1 Threat Model 22

2.1.2 System Design 23
2.1.3 Acquiring Program Variants 29
2.1.4 Limitations 30

3 Analysis **33**

3.1 Evaluation 33
3.1.1 Case Study: SQL Injection 34
3.1.2 Case Study: Varying microservice implementation 36
3.1.3 Case Study: Varying microservice version 40
3.1.4 Case Study: Variance Through ASLR 41
3.1.5 N-versioning components of GitLab 42
3.1.6 Performance 48
3.2 Discussion 56

4 Conclusion **57**

4.1 Conclusion 57
4.2 Future Work 57
4.2.1 Database Checkpointing 57
4.2.2 Alternative Implementations 58
4.2.3 Other Variation Mechanisms 58

A Additional Performance Plots **62**

B Source Code and Documentation **65**

List of Tables

3.1 Some vulnerabilities affecting GitLab components 45

3.2 Baselining network connectivity between server and client machines. 52

3.3 RDDR will be profiled in each of the above scenarios to learn the contribution of each component to performance. Only certain features will be enabled in each scenario, indiciated by check mark. 55

List of Figures

1.1 Attack surface reduction 6

2.1 RDDR block diagram. 23

3.1 Illustration of DVWA deployed with RDDR 35

3.2 RDDR denying access when divergent behavior is observed..... 36

3.3 Simplified view of GitLab architecture, borrowed from [1]. 43

3.4 Modified GitLab architecture with Postgres replicated behind RDDR... 47

3.5 Performance of RDDR normalized to the baseline for various different numbers of concurrent clients. Boxes span the 5^{th} through 95^{th} percentile 50

3.6 RDDR overhead normalized to baseline for 1 and 16 clients. 51

3.7 Throughput and latency for 10,000 transactions per client. 53

3.8 Throughput and latency for 10,000 transactions per client, normalized to each baseline..... 53

3.9 Aggregate CPU and memory usage for each deployment with 16 and 128 clients. 54

3.10 Analyzing the performance of RDDR with successive components removed. Each bar represents a different scenario enumerated in Table 3.3. 55

4.1 RDDR integrated into a service mesh as a sidecar of an n-versioned pod. 61

A.1 RDDR performance compared to baseline for 1 client. 63

A.2 RDDR performance compared to baseline for 2 clients..... 63
A.3 RDDR performance compared to baseline for 4 clients..... 63
A.4 RDDR performance compared to baseline for 8 clients..... 63
A.5 RDDR performance compared to baseline for 16 clients..... 64

Chapter 1

Background

1.1 Introduction

In the pursuit of reliable software, researchers have developed entire fields of study based on increasing reliability via replication and diversification of applications. These systems are studied in various forms in several research communities, including "moving target defense" (MTD), "service-oriented architectures" (SOA), and "n-versioned systems". For clarity, we will refer to any system where software is replicated, diversified, and checked for consistency as an "n-versioned system". N-versioned systems rely on diversity to catch vulnerabilities in computer systems. This diversity largely takes two forms:

1. Code diversity, via multi-programming or compiler-created variants (such as [2]).

2. System diversity, via architecture or operating system diversity.

We bring a practical implementation of n-versioning defense to distributed cloud applications with our system, RDDR. RDDR is an acronym that abbreviates

what our system does: **R**eplicate a request to N variants of a microservice, **D**e-noise non-deterministic behavior, **D**iff to find any differences in the instances' responses, and **R**espond with the appropriate reply. RDDR allows one to easily implement an n-versioned system for large-scale platforms that already use containers and container orchestration frameworks like Kubernetes [3]. See Figure 1.1 for an illustration of how RDDR can reduce the attack surface.

We avoid the need for manually generated diversity (such as prior work in multi-programming) by leveraging the fact that cloud microservices often follow a pattern of agile development and rapid deployment. We can derive diversity in part by deploying different releases of the same codebase. In this way, RDDR facilitates safer software upgrades. Consider a piece of software with an information leak vulnerability. Typically, a bugfix would be written, and the updated application would replace the previous version in production. Prior studies have shown that updates frequently cause further regressions [4,5]. With RDDR in place, both the patched and unpatched versions can be run in parallel and any bugs introduced by the patched version would be caught. This is because the behavior of the two application versions will constantly be compared to one another by RDDR. If either one's behavior diverges (either because the known information leak has been exploited, or a new one introduced by the patch has been triggered), RDDR will abort the connection and prevent the leaked information from reaching the attacker. In addition, we can n-version applications that use the same API but have different implementations (*e.g.* the PostgreSQL and CockroachDB databases). Software that share only an API and little in the way of source code should intuitively have highly disjoint attack surfaces. RDDR leverages this for increased robustness of the overall system.

Cloud applications these days often employ a distributed architecture com-

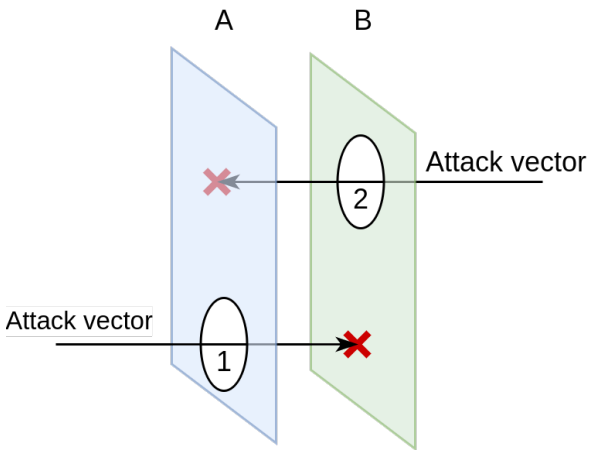


Figure 1.1: Above are two microservices A and B, with vulnerabilities indicated as "holes" in their attack surfaces. By deploying both microservices as an n-versioned set with RDDR, the attack surface is reduced to the set of vulnerabilities which are shared between the two, or in other words, where the holes in their attack surface overlap.

posed of many *microservices*. The design philosophy of such systems is to compose an application from simple services that work in unison to perform a complex function. GitLab is a real world example of an application designed according to this paradigm [6]. By distributing application functionality among a number of services, developers tend to achieve greater system robustness and scalability. Distributed applications are well-suited to n-versioning with RDDR, since developers can n-version just those microservices that are deemed critical to the operation of the overall system, thereby minimizing overhead. In a classic n-version system, this would be comparable to n-versioning critical segments of the codebase, as opposed to the entire application. To the best of our knowledge, this is not an application area where n-versioning has been practically applied. Prior work has explored methods of determining which services are most critical, for example: [7–9].

Although beyond the scope of this paper, RDDR could potentially be used to diversify instances of a microservice at the hardware level as well. Kubernetes can

orchestrate a deployment across multiple physically-diverse machines. The diversity of the system could stem from, for example, the ISA (x86, ARM, MIPS, *etc.*) or the chip manufacturer (Intel, AMD, Snapdragon, *etc.*). This use case is advantageous for a threat model where the hardware is the source of the vulnerability, as opposed to the software.

The remainder of this paper is formatted as follows: we begin by presenting the motivation for n-versioned systems in the cloud in Section 1.2. We then explore related work in Section 1.3 and describe how the lessons learned from these works contribute to the implementation of RDDR. A discussion of our system's design is documented in Section 2.1. Following RDDR's design, we evaluate both conceptual and real world use cases of RDDR in Section 3.1. We then finish with a discussion of RDDR in Section 3.2, our concluding remarks in Section 4.1, and a look at future work in Section 4.2.

1.2 Motivation

1.2.1 The Applicability of N-versioning to OWASP's Top 10 Web App Vulnerabilities

In this section, we study the most common vulnerabilities plaguing web applications today and investigate whether n-versioning can be applied to defend against each type of exploit. The types of vulnerabilities studied here are pulled directly from the OWASP Top 10 list of the most common vulnerabilities plaguing web applications [10].

#1: Injection

Injection involves an attacker injecting a procedure they control into the application and getting the application to execute it. This can be a crafted query, malicious server-side scripts, shell code injected in the stack, et cetera. Here we discuss ways in which each form of injection could be mitigated with n-versioning.

SQL injection: In this form of injection, an attacker is able to inject arbitrary SQL statements via an input form that the server will mistakenly execute on their behalf. This can be used to read and write privileged information, or corrupt the persistent storage of the application.

Usually, web applications feature some sort of sanitizer which treats user input so that it is not recognized as syntactic SQL. Some sanitizers can be thwarted, and presumably different sanitizers present different weaknesses. In theory, developers could n-version the SQL sanitizer used across N instances. An SQL injection may get past one, but not all, and this will lead to differences in behavior across the instances which an n-versioning monitor can detect and stop.

Another strategy involves replicating the database and microservices that use it and varying the names of the tables accessed by each microservice instance. A banking microservice might be interested in querying a table of credit cards, so it would look up the name of the credit card table in a mapping for that instance and find that the table name for this instance is `creditcards-89fds2`. A different instance's table would be named `creditcards-n25k9`. Essentially, the developer adds a level of indirection in the naming of tables. This should thwart many SQL injections, since the table name, injected by name by the attacker, will not be correct in every instance. One could implement this by first writing a script that appends a random string to every table in a database, and records the mapping from previous

name to new name in a file. One would then would then subclass the application's SQL client. The subclass would transparently map the table names in the user's query to the actual name of the table using the mapping we generated. This means no application code needs to change outside of instantiating the subclassed client.

Finally, one can also insert "honey pot fields" in the database tables that, if leaked, the n-versioning monitor will catch. These fields will be deliberately filled with random data that differ across instances. When an attacker goes to leak from the table, they are likely to carelessly dump all columns. The n-versioning monitor will see the "honey pot columns" differ across all instances and will detect the attack. This relies on SQL injections usually being catch-all `SELECT` queries that aim to leak whatever they can. The application will *never* read these fields during normal operation.

In our evaluation of RDDR in Section 3.1, we demonstrate a proof of concept n-versioned deployment which successfully closes an SQL injection vulnerability.

Code injection: There are still other forms of injection, such as code injection. A concrete example involves an attacker uploading a malicious PHP script to a server and getting the server to execute it. If the end result of the code injection is an information leak, naturally n-versioning is well-suited to prevent it as long as variants of the server code can be procured which differ in terms of their susceptibility to the malicious upload. Also see the discussion of the "Insecure Deserialization" vulnerability.

Buffer Overflow Injection: In this attack, a malicious user exploits a buffer overflow vulnerability to overwrite the return pointer in the server's stack and execute arbitrary code, including attacker-controlled code in certain cases. While many web applications are written in high-level languages that would not be susceptible to

buffer overflow, they may depend on lower-level libraries that are. ASLR is typically used to randomize the address space and make the exploit more difficult, yet it can be thwarted if a pointer is leaked to the attacker, thereby revealing the locations of various data in memory. In an n-versioned system with ASLR, however, if an attacker tries to leak a pointer, then the pointer will differ in every instance and the exploit will be caught. In our evaluation of RDDR, we demonstrate a proof of concept n-versioned deployment which leverages ASLR for protection against a pointer leak and buffer overflow injection.

#2: Broken Authentication

In this class of vulnerabilities, a web application's authentication and session management contain bugs allowing an attacker to compromise user secrets. For example, an attacker may aim to leak password hashes and salts from a database. The n-version monitor can catch the leak if each instance uses different salts to hash each password. This will be the case by default. When the instances report their salts, their responses will diverge and the exploit will be caught.

#3: Sensitive Data Exposure

This class of vulnerabilities covers a particular type of information leakage where sensitive data is transmitted in an unencrypted format and can be intercepted. This represents a configuration issue where data is leaked at all times during normal operation, as opposed to a bug where data is leaked due to a crafted exploit. The best mitigation is to properly protect data transmitted via every channel. Importantly, it's not enough to just encrypt the data in the database where it is stored, since it may be unencrypted when read out and sent elsewhere. This type of information

leakage is more difficult for an n-versioning system to protect against, since the leak is due not to a bug present in only a subset of variants, but to a configuration and/or application logic issue that would affect either all or none of the variants. Consider copies of a database deployed as an n-versioned set. The n-version monitor could catch the leak if a strict subset of the databases return the data unencrypted while the rest return the data encrypted, but it could not protect against the common mode failure where all return the data unencrypted.

If developers take care to architect their system such that trust is confined to one or a small number of microservices, then the trusted service(s) can be configured to properly encrypt data and then be n-versioned. In this case, any leaks of unencrypted data would be an anomaly that would be detected by an n-versioning monitor. In [11], Xia *et al.* present a similar architecture in the mobile domain, where trust is off-loaded to a security-oriented hardware unit called TinMan. They show that their system is effective at mitigating sensitive data exposure. Dynamic information flow tracking techniques could also be used to track the flow of unencrypted sensitive data through a distributed application.

#4: XML External Entities

This vulnerability is similar to an injection, but specifically affects XML parsers. The XML standard allows inclusion of external content in the XML document. An attacker can take advantage of this feature of XML to leak resources accessible to the server application, e.g. sensitive system files, available ports, or influence control flow e.g. execute remote code, or carry out a denial-of-service attack.

Essentially, external entities include content from other URLs into the XML document. This can trick the server into making a request to an arbitrary attacker-

provided URL. This can be used to probe the instance's view of the network, read sensitive files from the instance (using the `file://` protocol), etc.

We propose some defenses that use n-versioning to protect against leaking system files. System administrators can inject whitespace characters (or other uninterpreted characters) into system configuration files in each instance container that an attacker may try to leak. Functionally, these config files remain the same, but if they are printed out, the instances' traffic will differ and the leak will be caught. Additionally, administrators can randomize the port mapping in each instance container so that port scans will cause divergent behavior. Lastly, administrators can add randomly-generated user accounts to each container instance such that if /etc/passwd is ever leaked, behavior will diverge.

#5: Broken Access Control

This class of vulnerabilities involves an application failing to restrict a user's permissions to the intended set. As a result, an attacker may be granted access to resources which should be denied. The defense described above, where we vary the system config files in each instance, is also applicable here. If a sensitive file is not locked down as it should be, we can still detect whether it has been read by varying it across instances. In our evaluation of RDDR, we demonstrate two proof of concept n-versioned deployments which successfully close broken access control vulnerabilities in PostgreSQL.

#6: Security Misconfiguration

The authors cite this as the most common vulnerability. It is hard to prescribe a one-size-fits-all security policy to all developers since the policy usually depends on

the particular system topology. That said, one of RDDR's strengths is its ability to enable n-versioning regardless of system topology and detect a broad set of information leak vulnerabilities. With these properties, our system can provide enhanced security for a broad class of applications.

The authors also mention that verbose error messages can inadvertently leak info about the system that attackers can use for reconaissance. N-versioning can be applied to detect uncaught exceptions. Suppose instances of an n-versioned deployment are assigned a unique ID number. If this ID number were injected into every unhandled exception message (*e.g.* in a Python application), then instance behavior will diverge every time an unhandled exception is reached, which the n-versioning monitor can prevent from leaking.

#7: Cross-Site Scripting (XSS)

This is a client-side vulnerability where a malicious script is served to the user and executed by their browser. At this time, we are only considering n-versioned deployments on the server-side, so we consider this set of vulnerabilities out of scope.

#8: Insecure Deserialization

When an application suffers from this vulnerability, attacker-provided input is able to influence the server's state or control flow. An example is the billion laughs attack in which an untrusted input file is parsed leading to a denial of service [12]. In other cases, the attacker may be able to inject falsified data into a database, or trigger execution of gadgets in the server's application binary.

On the subject of denial of service attacks, if an attacker supplies an untrusted, malicious input file and the application parses it immediately, there is little oppor-

tunity to abort the attack using n-versioning. If the attack is successful and requires just one request, there will be no response from the N variants since they will have crashed upon processing the request. Were we to inspect application state across the N variants as well, it is challenging to deliberately cause variance when the exploit is underway. N-versioning alone is unable to patch denial of service attacks. However, prior work from Jones et al. has explored defeating denial-of-service attacks in n-versioned systems and could be extended to our system in the future [13].

On the other hand, n-versioning can be effective against data injection and control flow modification. If in an n-versioned deployment only a subset of instances are vulnerable to data injection, then n-versioning will be able to detect a difference in the traffic being sent to backend storage; some instances will send the data injected by the attacker, whereas the rest will not. Furthermore, if the vulnerability instead makes a subset of instances subject to control flow modification by the attacker, depending on the specific application the instances may behave differently in terms of the traffic they send back to the attacker. The n-versioning monitor will again catch this difference and abort the attack.

#9: Using Components with Known Vulnerabilities

In this vulnerability, developers expose themselves to exploitation by not patching known vulnerabilities in components of their system. To mitigate this with n-versioning, developers can deploy multiple instances of a component, each a different version. It's worth mentioning that this is a less "deliberate" method of varying your instances. That is, you are not *guaranteed* that instances' behavior will diverge in the presence of an attack. This is because the instances may all share the vulnerability and the exploit will continue to succeed because of common mode failure.

Also, components may exhibit vulnerabilities that wouldn't normally affect application-level software written for the cloud. For example, most web apps aren't vulnerable to buffer overflows by virtue of being written in safer languages. But they may depend on a library written in C that *does* have a buffer overflow vulnerability. If this is the case, variation via ASLR can also be useful for this class of vulnerabilities.

#10: Insufficient Logging and Monitoring

Insufficient monitoring of the system can make it difficult to know when an attack is underway or how one may have been carried out. N-versioning can be used to detect the occurrence of an attack and trigger verbose logging. N-versioning can also provide insight into the exact combination of inputs that led to divergent behavior.

Conclusion

In summary, n-versioning has the potential to protect against most entries in the OWASP Top 10 Vulnerabilities list. We note that it is less applicable to "Insufficient Logging and Monitoring", DoS attacks discussed in "Insecure Deserialization", and "Cross-Site Scripting". Ultimately, n-versioning's strength in this domain is in detecting behavioral anomalies among a set of microservice instances. This can be a difference in state, I/O, timing, etc. which could all be symptoms of any of the remaining categories of vulnerabilities discussed here.

1.2.2 Scenarios Where N-Versioning Can Be Beneficial to Cloud

There are many hypothetical web app deployments where n-versioning can pose a real benefit. Consider the following hypothetical scenario. A bug is discovered in a web application in production, and the development team sets off to quickly

release a patch for the vulnerability. However, in their rush to get the fix out, the developers introduce a new bug. Prior studies show that all too often software updates in fact lead to more bugs. A study by Crameri *et al.* investigated the hurdles to deploying updates in real systems [5]. The system administrators they surveyed reported that bugs in updates are one of the most common reasons why software updates fail. In another publication, Yin *et al.* found that between 14.8% and 24.4% of the OS updates they studied were implemented incorrectly [4]. RDDR can mitigate new vulnerabilities introduced in the latest releases of software. Simply deploy the current version and patched version alongside one another behind our proxy, and the system will be resilient both to the known vulnerability and to any new vulnerabilities that may have been introduced by the patch. Prior work by Hosek *et al.* studied an equivalent deployment strategy and found it to be effective at mitigating new bugs [14].

In another scenario, developers are unaware of any specific vulnerabilities that may plague their system that need to be patched, but they want assurance that their attack surface has been proactively minimized. By deploying multiple implementations of the same function alongside one another behind our proxy, the developers can be assured that their system is only vulnerable to the information leaks common to *both* implementations. There are a number of application types where diverse implementations are available. For SSH, there is OpenSSH and Dropbear; for databases, there is PostgreSQL, CockroachDB, and EnterpriseDB; for in-memory storage, there is Redis and memcached; for TCP/IP proxies, there is Nginx and Envoy; etc.

1.3 Related Work

Work related to RDDR touches various areas of computing including: Service Oriented Architectures, n-versioning, and moving target defense systems. There are many overlapping ideas and terms these paradigms share, however they are discrete enough to be logically separated when presented in relation to RDDR. RDDR borrows concepts from each area to create a system that can run in real time in a containerized environment via Docker.

The aim of these computation models varies depending on the research community that developed them. The SOA community typically focuses on improving quality of service (*e.g.* [15–17]). The n-versioning and MTD communities aims to mitigate vulnerabilities and improve software robustness (*e.g.* [18–20] for n-versioning and [8,21,22] for MTD). They each employ diversity in different ways to achieve these goals. We share their aim of mitigating vulnerabilities via diversity, though our approach to diversification is more similar to n-versioning. We borrow from the SOA community the distributed system architecture.

1.3.1 SOA

Service-oriented architectures (SOAs) have a long, rich history and distributed applications composed of microservices fall under the umbrella of SOAs. Many SOAs have proposed theoretical models which leverage containerization for redundancy and diversity. For example, in [17] Gorbenko *et al.* describe a system similar to RDDR's front-end proxy, which they dub "reliable concurrent execution." However, their system remains a theoretical model rather than a practical implementation like RDDR. In the course of implementing RDDR, we were forced to cope with many non-idealities

exhibited by real-world applications. These are described further in Section 2.1.2 on RDDR's design.

Diffy is a tool similar in design to RDDR, created by Twitter, used to perform regression testing in their SOA by running two versions of a single microservice (old and new) side by side [23]. This system was not developed to be run in production and was primarily developed to find "bugs in Apache Thrift and HTTP-based services." Diffy inspired the design of RDDR's incoming proxy, but RDDR builds on Diffy in several ways which enable it to be seamlessly integrated into distributed application architectures. Diffy only replicates traffic to the microservice replicas; it does not merge requests the replicas may make to downstream microservices. Were you to try to integrate Diffy into a production application, you would need to replicate *every* microservice downstream from Diffy. RDDR's outgoing proxy, on the other hand, is able to merge traffic streams, obviating the need to replicate any downstream services. Diffy also inspired RDDR's non-deterministic noise filter. That said, Diffy is unable to gracefully handle ephemeral state like CSRF tokens which it would naively filter out as non-deterministic noise. This makes Diffy unsuitable for many real-world applications which rely on CSRF tokens for security. In contrast, RDDR automatically identifies, saves, and later restores ephemeral state output by a microservice. These design decisions are explained at length in Section 2.1.2.

1.3.2 N-versioning

Work on n-versioning goes back decades and revolves around the idea of having multiple copies of a program/system (usually diverse) running in parallel to improve security. Early examples include redundant flight systems in aviation. N-versioning systems typically vote on responses generated by the different versions, with a *ma-*

jority rules schema.

In [18], Gholami et al. use n-versioning to reduce server load and improve performance of the distributed application. They vary service instances in terms of the number of features they offer. More lightweight instances can handle requests that don't require the full feature set. This is more efficient than scaling out the same service in equal numbers. Their architecture is similar to ours, but with a different goal in mind. Whereas they aim to increase efficiency of the deployment by varying the featureset, we aim to increase robustness by varying the vulnerability set.

Otterstad *et al.* describe in [19] the benefits of deploying diverse microservices behind a controller. Their system resembles RDDR insofar as they propose n-versioning microservices and blocking responses on divergence. They even mention how you could use the system like a honeypot, which we discussed earlier in Section 1.2.1. However, their proposal is merely a theoretical model rather than a practical implementation like RDDR. Like the work by Gorbenko *et al.* [17], they fall short of addressing the non-idealities of real-world applications.

1.3.3 MTD

MTD systems utilize diversity, however they typically vary applications over time rather than run diverse variants concurrently, like n-versioned systems. Prior works such as [8, 21, 22] apply MTD to distributed server applications. Torkura *et al.* use diversity over time in [8] to randomize the attack surface of cloud applications over time. They take a data-driven approach to diversifying microservices in order to mitigate high-risk vulnerabilities first. They do this by aggregating the known vulnerabilities a microservice is susceptible to and comparing their CVSS scores [24] and OWASP ratings [25].

In [22], Azab *et al.* present ESCAPE, wherein Linux containers are migrated from host to host on-demand to evade attackers. A monitor looks for anomalous behavior from the containers, which is assumed to indicate an attack underway, and will initiate a migration on sight. Their evaluation consists of a simulation of a representative mathematical model, but the system proposed is never actually implemented.

1.3.4 N-versioning and MTD in Other Domains

So far, we have discussed related work in the same domain as RDDR: distributed cloud applications. However, these fields have a broad reach, touching a number of other domains from operating systems to software-defined networking.

In [20], Österlund et al. use n-versioning (also known as multi-variant execution) to improve robustness of the Linux kernel. Like us, the authors aim to protect against *information leaks* by deploying two diverse kernels in parallel. They rely on a *variant generator* to manually create variance among the kernels so as to cause divergence on information leaks. Contrast this with systems where diversity is derived more organically, such as by multiple teams implementing the same specification in isolation [26–28]. For our evaluation, we study deployments of RDDR where diversity is sourced organically from different versions and implementations of software, and automatically from the OS.

Software-defined networking has been used to implement moving target defense in [29]. The authors argue that a variable software-defined network is a natural fit for moving target defense where the computing environment, namely the network in this case, is constantly adapting to attackers' strategies. They report success in delaying an attacker's discovery of critical network services, i.e. the reconaissance phase of an attack.

1.3.5 Currently Available Technology

Production-level proxies like Envoy [30] and Nginx [31] offer a feature called "traffic shadowing", which is comparable to RDDR, but falls short of RDDR's level of protection against attacks. In "traffic shadowing", the traffic to a microservice in production can be replicated and forwarded to a "shadow" copy. This can help developers regression test updates in development with real-world traffic prior to deploying them. While this technique can help to discover vulnerabilities, it doesn't add any protection to the production microservice. The proxy will ignore all traffic returned by the shadowed instance rather than derive useful information from it. N-versioning with RDDR, on the other hand, actually sits on the critical path of traffic and has the potential to shrink your deployment's attack surface because the shadow instance's traffic is actively monitored for consistency with the other instance(s). On top of monitoring traffic, RDDR provides features that are necessary when deployed in real-world applications that exhibit random noise and instance-specific state, which a shadow proxy could not handle by itself.

And RDDR is not just a shadow proxy with traffic differencing; it also provides features which are necessary for use with real-world applications that exhibit random noise and instance-specific state, which a shadow proxy alone cannot handle.

Chapter 2

Our System

2.1 Design

In Section 1.2, we detailed numerous scenarios where n-versioning could be deployed to address the most common vulnerabilities in cloud applications. We then discussed recent work on n-versioning and similar systems in Section 1.3. In that section, we highlighted certain theoretical models of n-versioning applied to cloud platforms ([18, 19]). In this section, we present the design of our system, RDDR. It builds on the models presented in prior work and makes several contributions which are crucial for practical deployments on real-world applications.

2.1.1 Threat Model

RDDR aims to make distributed applications more resilient to remote attacks. The attacker aims to leak data from the distributed application by leveraging one or more vulnerabilities in the application's constituent microservices. It is assumed that the data will be leaked through one of the microservice endpoints, rather than through a side channel. [32–39] are a few exemplary vulnerabilities of this variety. To protect

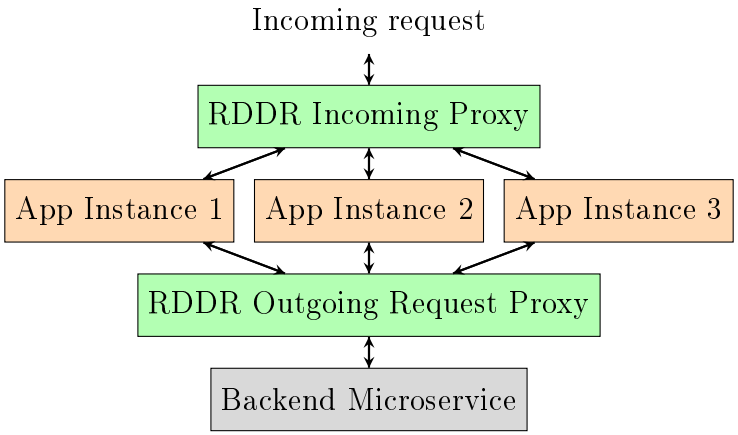


Figure 2.1: RDDR block diagram.

against side channel leakage, we defer to these prior works: [40–42]. If the information leak can be automatically detected and prevented, even if this means returning no data to the attacker, we consider the vulnerability mitigated. The attacker is assumed to be unprivileged in the context of host system administration and in the context of the web application.

2.1.2 System Design

Our system is called RDDR, which stands for Replicate, De-noise, Diff, and Respond. These are the main steps our system takes in order to detect divergent behavior. First, critical microservices are **replicated** and diversified across several instances. Incoming traffic is then **replicated** and sent to each instance. The instances send their responses. RDDR **diffs** the responses to detect divergent behavior, and if none is detected, it will **respond** to the client. This is a high-level description of how the system works. Later on, we will describe subtleties of our system such as how we handle non-deterministic noise and ephemeral state.

As stated in Section 1.1, RDDR helps to achieve application robustness with

lower overhead by exploiting the distributed nature of cloud applications. Developers need only create copies of critical microservices and manage them using RDDR rather than make copies of the entire application, which would be more costly. Furthermore, modern container orchestration tools like Kubernetes [3] and Docker Swarm [43] simplify the process of replicating your microservice from a base image. All of this pre-existing infrastructure, and the fact that RDDR itself can be run as a container using a Docker image, make RDDR straightforward to deploy.

RDDR is written in Python 3 with support from the asyncio event loop library. The source code is freely available online under the MIT License [44]. Architecturally, RDDR can be visualized as a set of proxies on either side of the N variants of the critical application. Both of these proxies operate at the transport/socket layer. Each proxy binds to an IP and one or more ports and waits for incoming connections. Depending on the type of proxy, incoming data will be processed differently before being forwarded to the next endpoint.

One proxy is deployed to handle incoming requests to the variants. This is the **RDDR Incoming Proxy** shown in Figure 2.1. It will replicate and make any necessary modifications to traffic coming into the proxy and then forward the traffic to all variants. Modifications made to inbound traffic are described further in Section 2.1.2. The Incoming Proxy will compare any traffic sent by the variants in response to check for divergent behavior. The traffic is tokenized and differenced according to the messaging protocol. If any meaningful differences are seen, excluding non-deterministic noise, the proxy will close the connection to the client and forward them none of the data. Aborting communication like this helps to avoid information leaks experienced by only a subset of the instances. See Section 2.1.2 for more on how non-deterministic noise is allowed to propagate through the proxy.

Deployed "behind" the application variants, there may be zero or more of the **RDDR Outgoing Proxy** as seen in Figure 2.1. This proxy performs a similar function to the Incoming Proxy, but it monitors traffic between the microservice variants and one other microservice in the backend. This enables catching information leaks that may occur on a network connection other than the one that is directly client-facing, which the Incoming Proxy handles. The Outgoing Proxy's functionality is a mirror image of the Incoming Proxy, in that it differences messaging coming from the N variants, rather than messages going to them. Additionally, it modifies and replicates the response to their query. Lastly, the Outgoing Proxy allows non-deterministic noise through just like the Incoming Proxy. The Outgoing Proxy differs from the Incoming Proxy in that it exposes a different port for each of the microservice variants to connect to. Each of the variants is expected to connect to a specific port on the Outgoing proxy so that the proxy can distinguish which traffic was sent by which variant. Both the Incoming and Outgoing Proxies are started from the same binary and share a container.

Protocol Support

RDDR supports various transport and application layer protocols. Presently, it supports unencrypted TCP as well as encrypted SSL at the transport layer. It also ships with support for some application layer protocols including PostgreSQL, HTTP, and JSON. Support for application layer protocols is implemented by separate Python modules with a standardized interface, allowing developers to extend RDDR to support new protocols as desired. These modules handle all protocol-specific tasks such as tokenizing, differencing traffic, and modifying traffic. Additionally, a generic bytewise differencing module is provided for strict, general purpose monitoring.

The application protocol modules tokenize and difference the traffic according to the semantics of the protocol. For example, the HTTP module tokenizes at the newline boundary and diffs lines against one another. The HTTP module will also interpret the HTTP header and decompress the message before diffing if necessary, as well as save CSRF tokens; more on this later. The PostgreSQL module tokenizes the traffic into separate messages according to the PostgreSQL message format [45] and will diff messages of types deemed critical.

Handling Nondeterminism

In Section 1.3, we discuss some of the related work on theoretical models for n-versioning in the cloud. These models share some aspects of our design, but lack others. One feature lacking from these models, but sorely needed for practical implementation, is how to distinguish random noise from real divergent behavior.

Suppose that the N microservice instances send a random string to the client. A real-world example is PHP session IDs, which web applications use to uniquely identify users. Because each instance generates a different random string, their traffic will appear divergent even in the absence of a bug. To distinguish random noise from real bugs, we have implemented filtering similar to Diffy's approach [23]. For n-version deployments that exhibit some random noise, one can deploy two instances of the same microservice (no diversity applied) and use them in combination to determine which divergences are random and which are due to the diversity. RDDR is programmed to filter based on the traffic received from these two instances which comprise the so-called "filter pair". In the case of generating random strings, even the two identical instances will differ in the string they produce. We know that these instances are identical and thus, this particular divergence cannot be due to any bug present in the

application. With filtering turned on, RDDR will only categorize traffic divergence as a bug if all instances *except* the "filter pair" differ from one another.

We emphasize that the filter pair **must** behave differently from one another in every instance of non-deterministic behavior. Therefore, developers will want to make sure that they are using a cryptographically-secure source of randomness to avoid duplicate values. Similarly, developers should take care when reducing the entropy of a random variable, for example by choosing a random integer from a small range where the probability of duplicate values is higher.

Handling Ephemeral State

Another feature lacking from theoretical models but required for real deployments is a mechanism to accomodate ephemeral microservice state used in client-server handshakes. Consider the case of CSRF tokens. These are randomly generated strings which the server embeds in HTML forms requested by clients. The server stores these tokens and checks to make sure that each request submitted has a valid token, i.e. that it corresponds to a real form that was previously requested. This mitigates cross-site request forgery (CSRF), where a user may be tricked into submitting POST data to the server by means other than the official HTML form, such as by clicking a malicious link. If CSRF tokens are used by a microservice that is n-versioned using a more naive system, the tokens will appear to the n-version monitor as non-deterministic noise and be filtered as previously described. The monitor would then forward one of these tokens on to the client as part of the HTML form. However, a problem arises when the user subsequently submits the form with the token embedded in it. The user's token is valid for only one of the N instances, yet it will be replicated to all. Therefore, only one instance will properly handle the

user's request; the others will reject their token and the monitor will think it has seen a bug when comparing their responses. To remedy this, RDDR may make minimal modifications to the traffic being sent to the N instances so that each receives the correct token.

To do this, RDDR's HTTP handler plugin actually stores values it suspects are CSRF tokens and reinserts them later. As the plugin scans the traffic being sent to the client, it will not only ignore lines that differ across all instances. Within that line, it will look for the character ranges that differ, and if all are alphanumeric and at least ten characters, the plugin will save those strings. The criteria for saving—alphanumeric and at least ten characters—is fairly arbitrary and seemed to work in practice with our CSRF tokens. The plugin remembers which CSRF token was sent to the client, and which CSRF token each microservice instance expects to see later. When the plugin observes the user's CSRF token in a later request, it will substitute his or her token with the one appropriate for each microservice before sending it along. After forwarding, the set of CSRF tokens can be removed from memory.

As of now, the HTTP extension is the only one which stores tokens in this fashion. However, the ability to modify traffic is included in the standard plugin API, so any developer is free to implement a similar solution should their protocol require it.

Handling Known Variance

Some application variants will inevitably behave differently. Consider different versions of the same application deployed behind RDDR of which you can request the software version. Upon requesting the version, RDDR is likely to identify deterministic divergence, since the version string depends directly on the version of the

program. To mitigate this phenomenon, you can statically configure RDDR to ignore foreseen divergence. This is currently only implemented for the Postgres plugin but could be extended to others.

2.1.3 Acquiring Program Variants

Because RDDR requires unanimous agreement among the microservice variants for a response to go through, the information leak attack surface of the system effectively becomes the intersection of the attack surfaces of all instances. Therefore, RDDR is only able to shrink a system's attack surface if the constituent microservice instances are diverse. There is an additional requirement that though instances are diverse, they behave the same way with regard to some specification. This enables RDDR to pass along traffic adhering to that spec, which will be the same across all instances. For example, SSH servers from different developers are expected to both adhere to the SSH protocol. There are a number of ways developers can achieve diversity among the N variants they deploy, and we discuss some techniques here.

Developers may deploy **diverse implementations** of a particular microservice function from different vendors. When we say function, we mean a logical component of a system such as a database or proxy that performs a well-specified function. Databases like PostgreSQL, for example, implement a well-specified query language and network protocol. Other database technologies exist which conform to the same query language and protocol, like CockroachDB [46] and EnterpriseDB [47]. Because these three applications adhere to the same network protocol and ought to behave the same given the same inputs, it is straightforward to deploy them as diverse instances of the same logical database.

A different technique involves deploying **different versions** of an application

from a shared codebase. This technique can be helpful in closing vulnerabilities in an old version of the microservice while simultaneously preventing exploitation of new bugs introduced by a software update. An important stipulation is that these versions should share at least the set of features that are required to implement the overall application. This technique was explored by Hosek et al. in [14], who found that new vulnerabilities introduced by software updates could be successfully mitigated by deploying alongside older versions.

There are also automated ways of generating diversity that don't incur software development overhead. One way is to leverage the operating system. The OS can diversify software at the process level by randomizing the address space, inserting randomized stack guards, and using disjoint code layouts. Prior works such as [48–50] have explored n-versioning implementations relying on these techniques. Developers can also leverage tools like the compiler to introduce diversity. [51] provides a useful overview of state-of-the-art techniques for automating software diversity, including mutating and reordering basic blocks, randomizing the stack layout, and inserting garbage code, among many other techniques. [52] leverages compiler-based diversification to increase software robustness to transient hardware errors. AVATARs, a project from NEC Laboratories, uses a combination of diverse compilers and source-to-source translators to yield diverse executables from a single codebase [53].

2.1.4 Limitations

Our solution potentially makes the system more vulnerable to denial-of-service attacks. Consider the case where we have 3 different versions of an app deployed, and one of them has a vulnerability that can cause runaway CPU utilization and thus denial of service. The n-versioned system will still be vulnerable to denial of service

since the buggy app will receive the malicious input and begin hogging resources. While out of scope for this paper, denial of service could be mitigated by augmenting RDDR with a timeout counter. If an instance takes longer than the timeout to respond, it will be restarted. We refer to [13] for ways to combat an attacker who repeatedly triggers a denial of service in an n-versioned system.

Our system potentially broadens the timing attack surface. Consider application A deployed alongside application B. A contains no timing bugs, whereas B contains one timing bug that can leak sensitive information: B will take longer to respond to a login request with a valid username than one with an invalid username. If A and B are deployed side by side and RDDR waits for them both to respond, the timing channel in B is still exploitable. Prior work from Yin et al. explores how to protect against attackers exploiting the weakest instance in an n-versioned deployment to learn information via a side channel [54].

N-versioning would not be applicable for services that generate instance-specific secrets, such as multi-factor authentication. This is OK; you would simply factor out the multi-factor auth service into its own service. Multi-factor auth is not likely to be very vulnerable since it does not directly receive user input. If an application were to generate secrets which flowed (directly or indirectly) to the client, then the de-noising step of our system would eliminate the possibility of a false positive.

We do not consider vulnerabilities that may exist in the RDDR proxies themselves. It is possible our software is imperfect and introduces new bugs. Our software would need to be subject to testing and scrutiny for developers to feel comfortable deploying it. We have decided to open source the project in part to encourage community members to audit the codebase.

There is the possibility that RDDR may register a false positive if microser-

vices contain time-varying information in their outputs. Consider a microservice that reports a coarse-grained timestamp. If a request is made on a time boundary to a 2-version deployment with non-deterministic filtering, there is a chance that the filter pair both report time as "12:10", whereas the third instance reports the time as "12:11". RDDR would detect divergent behavior and block the response, a false alarm. In our deployments, however, we have found this not to be an issue.

Some forms of ephemeral state will not be able to be reinserted by RDDR. Consider the case where the application issues a challenge to the user which differs from one instance to the next. The user is expected to process the challenge string in some way (perhaps by signing it with a private key) and return the result to the application. RDDR will save the various challenges from each application instance, but because the user modifies it on their end, RDDR will be incapable of inserting the appropriate response to each instance. To remedy this, we recommend leveraging the microservice architecture and factoring out the generation of the challenge string into a separate, single microservice. All application instances will make an outgoing request through the Outgoing Proxy to this microservice and as a result, they will all receive the same challenge string. This way, there is nothing RDDR needs to modify.

Chapter 3

Analysis

3.1 Evaluation

First we will apply RDDR in a number of different deployments and study the vulnerabilities it is able to close. All deployments were carried out on real hardware, and the Kubernetes files implementing each deployment are available in the RDDR source code repository. By showing that RDDR patches known vulnerabilities, it is reasonable to extrapolate that RDDR should also be able to patch similar, undiscovered vulnerabilities in the future. We will demonstrate diversifying instances of a microservice by varying: the version, the implementation, and the address space of that microservice. After analyzing some small-scale case studies, we will demonstrate RDDR deployed within GitLab, a large-scale distributed application which already employs many industry best practices for security. We conclude by measuring the overhead incurred by n-versioning with RDDR.

3.1.1 Case Study: SQL Injection

The Damn Vulnerable Web App (DVWA) is a website that demoes many commonplace web vulnerabilities [55]. It was developed to help system admins and web developers hone their understanding of web security. DVWA can be configured for different security levels to make the job of exploiting vulnerabilities more and more difficult. One such vulnerability is SQL injection. In an SQL injection, an attacker is able to modify the function of a benign query in order to perform a malicious exploit. Often this is accomplished by escaping the quotation marks where user input is placed, causing the user input to be interpreted as actual SQL syntax. This enables attackers to read the contents of privileged tables and even make modifications to the database. Query sanitization is the first line of defense against SQL injection. By sanitizing the query, the application can detect and prevent characters of the user's input like apostrophes and quotation marks from being interpreted as SQL syntax. Different DVWA security levels will sanitize user input to varying degrees.

With RDDR, we can harden DVWA against SQL injections. We have modified DVWA slightly so that it uses an external database rather than one built into the container. From there, we deploy three instances of the DVWA frontend communicating with a single backend database through RDDR's outgoing proxy. An illustration of the deployment topology is shown in Figure 3.1. The DVWA instances are configured with different levels of security so that they sanitize user input differently. One instance is configured for greater input sanitization, whereas the other two instances, forming the filter pair, don't do any input sanitization. This should lead to divergent behavior when an SQL injection is underway which RDDR can detect and stop. This case study emphasizes the necessity of the outgoing proxy. The outgoing proxy enables our N instances to communicate with a single backend database and is

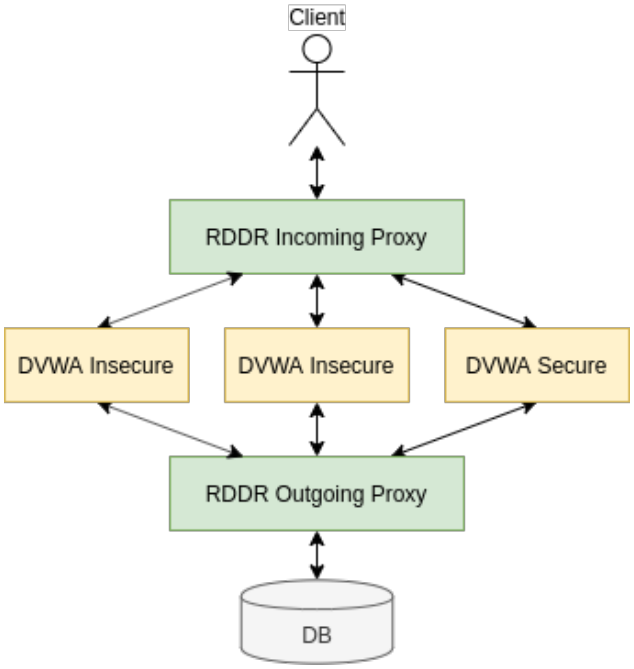


Figure 3.1: Illustration of DVWA deployed with RDDR

instrumental in detecting the divergence in the queries sent to it.

The deployment is successful at preventing SQL injection and letting through benign traffic. When the user requests the SQL Injection demo page, their request is forwarded by RDDR's incoming proxy to every instance of DVWA's frontend. The instances each send the webpage, which contains an input form and CSRF token. As described in Section 2.1.2, RDDR will automatically identify the CSRF tokens and save them. The user is forwarded the page sent by the first instance. Consider an attacker who tries to exploit the form and insert an SQL injection which reads passwords from the database. The attacker will submit the form as a POST request back to the cluster. RDDR will replicate the user's request for each instance and substitute the CSRF token it previously saved with the right token for that instance. Then it forwards the traffic to each instance. To respond to the user's POST request, each frontend instance must request data from the database to present to the user.

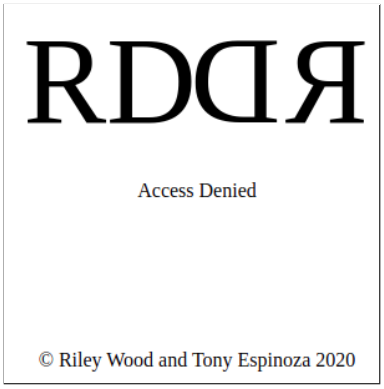


Figure 3.2: RDDR denying access when divergent behavior is observed

They insert the user input into a query template; the insecure instances do not sanitize the query, whereas the secure instance does. The instances send their MySQL queries to RDDR's outgoing proxy. RDDR gathers the requests from every instance and compares them for equality before merging them and sending one request along to the actual database. It is here that the outgoing proxy detects the exploit. It will see that the secure instance has escaped the quotation marks input by the attacker while the other two instances have not. Upon seeing this divergence, the outgoing proxy will terminate the socket opened to the frontend instances, and they will subsequently return an HTTP error code. These HTTP error codes happen to also be flagged as divergent by the incoming proxy, which renders the denial message to the attacker presented in Figure 3.2. When users make benign requests using the form, no divergence is detected by the outgoing proxy, since no syntactic characters are inserted by the user.

3.1.2 Case Study: Varying microservice implementation

As mentioned in Section 2.1.3, developers can diversify instances of an application by sourcing implementations from different development teams. For certain

applications, there exist a number of vendors offering products that comply with the same interface. We note that for RDDR to be applicable, **all application instances must implement the same application layer protocol (e.g. HTTP)**, or else a translation layer is needed so that all benign traffic looks identical to RDDR across all instances. CockroachDB and PostgreSQL are two databases that use the same network protocol. PostgreSQL was developed first in 1996 along with its protocol, known as `pgwire`. CockroachDB, created at Cockroach Labs and first released in 2017, was designed to comply with the PostgreSQL network protocol so as to be compatible with existing tools and infrastructure. This means the two can be deployed together behind RDDR without the need for a translation layer. In this case study, we show that a configuration consisting of two Postgres instances and one CockroachDB instance accepts benign traffic and is able to detect and prevent an information leak vulnerability affecting Postgres.

```
1 CREATE FUNCTION leak2(integer,integer) RETURNS boolean
2     AS $$BEGIN RAISE NOTICE 'leak %%', $1, $2;
3         RETURN $1 > $2; END$$
4     LANGUAGE plpgsql immutable;
5 CREATE OPERATOR >>> (procedure=leak2, leftarg=integer, rightarg=
    integer, restrict=scalargtsel);
6 SET client_min_messages TO 'notice';
7 EXPLAIN (COSTS OFF) SELECT * FROM some_table WHERE col_to_leak
    >>> 0;
```

Listing 3.1: Exploit for CVE-2017-7484

The vulnerability we will mitigate is CVE-2017-7484, an information leak in Postgres up to version 9.2.20 [56]. The vulnerability is due to a bug in how Postgres enforces access control when doing query planning. An attacker leverages a custom-

defined function and operator to leak information via a `SELECT` query. While protected data won't be leaked in the query results themselves, it *is* leaked when the database creates a query plan, which *will* pass sensitive information to the attacker's custom operator. To be vulnerable, a protected table must exist within the database. This is any table that one or more users are unable to read from. An unprivileged user can use the exploit shown in Listing 3.1 to create a custom operator which leaks this table during query planning despite it ostensibly being protected.

Key to our n-versioned deployment is the observation that this vulnerability only affects Postgres, and not CockroachDB. CockroachDB is inherently immune to this CVE, since it does not support user-defined functions and operators. We should note that if these features are a necessity for the overall application, this would not be a suitable deployment. All instances that make up an n-versioned deployment must share the minimum set of features needed to implement the target application. Otherwise, developers will encounter divergent behavior when one instance executes the feature and another raises an error. In addition, Postgres and CockroachDB need to be configured to behave as identically as possible. For example, in our deployment we had to manually configure Postgres' transaction isolation level. CockroachDB forces serializable isolation (the strictest setting), so Postgres must do the same. There are certain cases where the two implementations cannot be made to behave the same, such as when they report their version strings. CockroachDB will of course report "CockroachDB" and its version, whereas Postgres will report something different. Thankfully, RDDR can be pre-configured to ignore any anticipated, benign divergence. For our deployment, we configured RDDR to anticipate the different version strings. Upon seeing the version string for a given instance, RDDR will substitute it with the same string for every instance before differencing. This does not change

the traffic received by the end client. We also found that CockroachDB and Postgres differed in how they ordered rows of query results. Because the PostgreSQL query language does not guarantee any particular row order unless specified by the `ORDER BY` keyword, each implementation is allowed to order rows however it sees fit. Naturally, when they differ RDDR blocks the connection, despite the traffic being benign. To avoid the issue, developers would need to explicitly order all queries.

Once the N variants are appropriately configured, CVE-2017-7484 can be mitigated. The exploit shown in Listing 3.1 will fail at the very first step. Whereas the Postgres instance will indicate successful creation of the custom function, CockroachDB will raise an error indicating that the feature is not supported. RDDR will identify the difference in traffic and break the connection before either response can reach the client. The attacker can try to reconnect and proceed with subsequent steps of the attack, but the final `EXPLAIN` query which causes the leak will always be blocked. The Postgres instances, as we know, are vulnerable to this attack and will in fact still leak information from the protected table. But CockroachDB, having failed to create the custom operator and function, will throw an error complaining that the `<<` operator cannot be parsed. Because RDDR will again identify the difference in their responses as divergent behavior indicative of a leak, CockroachDB's behavior helps to protect PostgreSQL from leaking sensitive information.

This deployment demonstrates the feasibility of deploying very diverse implementations of the same logical function behind RDDR for reliability. It also illustrates some of the difficulties developers may encounter, such as having to circumvent unspecified behavior like row order. However, once these issues are sorted out, this kind of deployment is very powerful. Like any RDDR deployment, this one is only susceptible to information leaks that plague all instances, i.e. common mode failures. Prior

work from the Project on Diverse Software (PODS) found that common mode failures in diverse software implementations are rare, and the few that crop up are generally due to a flaw in the specification itself rather than a recurrent bug present in every codebase [57]. This suggests that an RDDR deployment of diverse implementations such as PostgreSQL and CockroachDB should be especially resilient to information leak vulnerabilities.

3.1.3 Case Study: Varying microservice version

In this case study, we explore an n-versioned deployment where diversity is derived from deploying different versions of the same microservice. The application to be studied is Nginx, a widely used proxy for web applications. A recent vulnerability, CVE-2017-7529 [32], demonstrated that Nginx up to version 1.13.2 was vulnerable to an integer overflow which could leak sensitive server information. The vulnerability is due to Nginx processing a content range request incorrectly. The attacker sends an HTTP request with a crafted Range header. Nginx fails to check the bounds of the user-supplied range, and experiences integer overflow when calculating the size of the payload to send back. It calculates a size larger than the requested document itself, and ends up sending data past the end of the document to the client. The result is a very long response sent back to the client containing the document concatenated with the contents of server memory.

We are able to patch this vulnerability using RDDR deployed in front of multiple versions of the Nginx proxy. We deploy a 3-version configuration of Nginx, where the two instances comprising the filter pair are running version 1.13.2, and the third instance is running 1.13.4 which does not exhibit this vulnerability. This models a hypothetical deployment strategy where developers deploy old and new versions of

the same application in parallel behind RDDR in order to patch bugs in the old version as well as prevent exploitation of any new bugs that may have been introduced by the patch. We carry out the exploit against our n-versioned Nginx deployment and find that RDDR successfully aborts the connection. It sees that one instance's response is much longer than the other's which it treats as divergent behavior. With that, the information leak has been patched.

This case study emphasizes the applicability of RDDR to deployments where diversity is achieved by deploying different versions of the same application. We also want to emphasize how easy it is to construct an n-versioned deployment this way. Containerization platforms like Docker have built in support for specifying image versions by tag, so it is straightforward to spin up the exact versions of a container that you need.

3.1.4 Case Study: Variance Through ASLR

Buffer overflows are a common avenue for attackers to begin an exploit. It used to be that a buffer overflow could directly enable hijacking control flow by overwriting the stack's return address with the location of the exploit code. This can be attacker-controlled code such as shell code injected in the overflowed buffer itself, or a gadget that's already present in the application binary and can be used maliciously by the attacker [58–60]. Nowadays, with the widespread adoption of address space layout randomization (ASLR), buffer overflow attacks are made more difficult. Because the operating system varies the virtual address space each time the program is executed, the attacker no longer knows where their exploit code is located. Before the attacker can hijack the program, he or she must learn the address of the code to jump to. This can be done by causing the program to leak a pointer to the section of memory of

interest.

RDDR can play a useful role in preventing pointer leaks in the presence of ASLR. We have developed a C program that is vulnerable to a buffer overflow attack. The program is a simple echo server that stores the requester's message in a buffer and sends it back. However, the program does not check for overflow, so if the requester overwrites the null terminator at the end of the buffer, the program will leak a pointer adjacent to the buffer in the stack. When the program is deployed with ASLR, this helps the attacker learn the location of a gadget to exploit. An attacker's exploit would proceed as follows: first, send a large payload to cause the program to leak a pointer; second, calculate the address of the gadget as an offset from the leaked pointer; third, send an even larger payload to overwrite the return address with the calculated address of the gadget. RDDR is able to stop the exploit at step one by detecting and preventing the pointer leak. When two instances of the same binary with ASLR are n-versioned, each will have a unique address space. When the attacker tries to leak a pointer, each instance will report a different address. RDDR will treat this as divergent behavior and close the connection to the attacker. This deployment does not use non-deterministic filtering, since RDDR's filter would choose to ignore the addresses as they would differ across every instance including the filter pair.

3.1.5 N-versioning components of GitLab

Here we apply RDDR to a deployment of GitLab [6], a web-based platform for hosting and collaborating on source code repositories. We will show that n-versioning with RDDR need only be applied to a subset of the containers in the deployment to achieve resilience.

Next we will describe the architecture of GitLab, a distributed cloud applica-

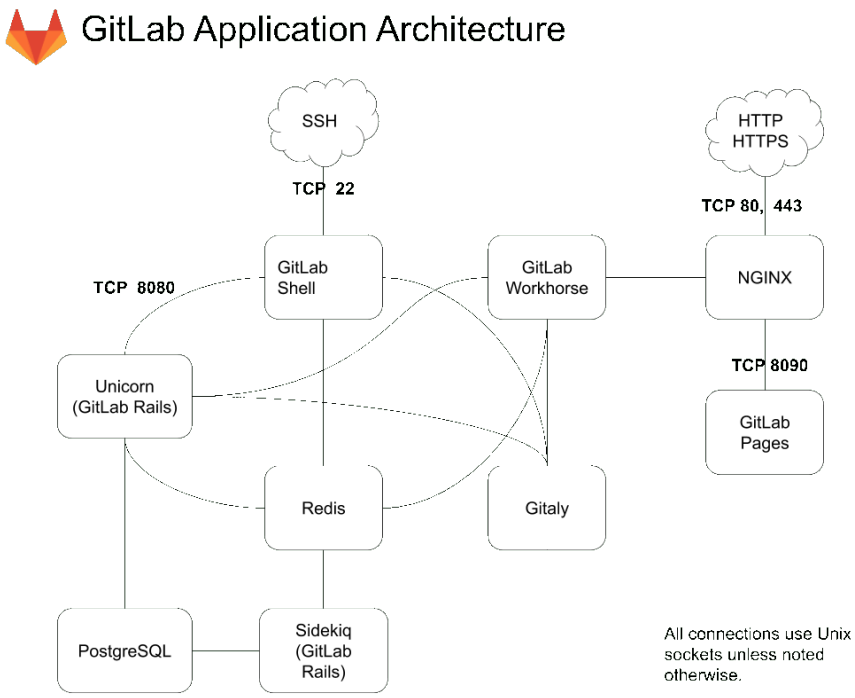


Figure 3.3: Simplified view of GitLab architecture, borrowed from [1].

tion which exemplifies many best practices in cloud security. We study the microservices that make up the architecture, vulnerabilities associated with those components, and ways in which n-versioning may provide resilience to those vulnerabilities.

GitLab Architecture

GitLab is a web-based platform enabling individuals and teams to host and collaborate on source code repositories. It also has facilities for hosting web pages like project documentation, deploying continuous integration pipelines, collecting project analytics, and tracking bugs. The application is constructed from a number of smaller microservices, some of which are developed in-house by the GitLab team and others which are open source projects of their own. Examples of the latter include GitLab's Postgres database and Nginx ingress proxy.

Figure 3.3 presents a simplified view of the GitLab architecure. At the top of the diagram are client interfaces for SSH and HTTP(S). These traffic are routed to GitLab shell and the Nginx ingress proxy respectively. From there, the request may be passed to a number of services depending on the nature of the request. For example, requests of dynamic GitLab content will be routed to the Unicorn web server, whereas requests for static content will be handled by the GitLab Pages module.

For this study, we are concerned with vulnerabilities that manifest themselves in a particular microservice and therefore may affect GitLab when deployed. Table 3.1 enumerates a subset of vulnerabilities affecting GitLab components from the last few years.

Component	Description	CVE	Vulnerability
GitLab Runner	Executes GitLab CI jobs	CVE-2019-11000	Group runner registration token disclosure
		CVE-2019-7549	Unauthorized users can view others' job info.
		CVE-2017-0918	Remote code execution
GitLab Workhorse	Offloads compute	CVE-2018-19583	Sensitive information visible in log files.
GitLab Geo	Geographically distributed GitLab nodes	CVE-2019-10117	Open redirect issue
Gitaly	RPC Server	CVE-2019-11549	Info leak: HTTP/GIT credentials are included in logs on connection errors
GitLab Pages	Hosts static websites	CVE-2018-19572	Unauthorized access to files in chroot environment
		CVE-2019-6783	Directory traversal allowing remote code execution
		CVE-2019-5467	Persistent cross-site scripting
PostgreSQL	Database	CVE-2017-7484	Data can be leaked from privileged tables using client-defined operator
		CVE-2019-10130	Data can be leaked from privileged table rows using client-defined operator
		CVE-2019-10164	Buffer overflow when altering user password
Nginx	Ingress proxy	CVE-2017-7529	Integer overflow can leak server memory.

Table 3.1: Some vulnerabilities affecting GitLab components

N-versioning Postgres within GitLab

We previously showed that RDDR could be used to secure Postgres against CVE-2017-7484, an information leak bug. In GitLab, we will again secure Postgres, this time against a different info leak: CVE-2019-10130. We will apply RDDR to the Postgres database microservice in a 3-instance configuration to mitigate this vulnerability.

```
1  -- Create leaky function, and operator to call it
2  CREATE FUNCTION op_leak(int, int) RETURNS bool
3      AS 'BEGIN
4          RAISE NOTICE ''leak %, %'', $1, $2;
5          RETURN $1 < $2; END'
6      LANGUAGE plpgsql;
7  CREATE OPERATOR <<< (procedure=op_leak, leftarg=int, rightarg=int
       , restrict=scalarltsel);
8  -- Will call leaky function, printing rows to console.
9  SELECT * FROM some_table WHERE col_to_leak <<< 1000;
```

Listing 3.2: Exploit for CVE-2019-10130

CVE-2019-10130 is an information leak affecting Postgres 10.x up to 10.7. This vulnerability allows a user-defined operator to leak privileged information from a table enforcing per-row security policies. It is one of two recent vulnerabilities that arose due to bugs in how Postgres enforces access control on statistics views [56,61]. For this investigation, we assume there exists an SQL injection vulnerability in the frontend of the application enabling an attacker to send arbitrary SQL queries to the backend database and carry out the exploit. To be vulnerable, first a privileged database user must create a table with row-level security, and then grant `SELECT` privileges to another database user, but deny access to one or more rows. The unprivileged user

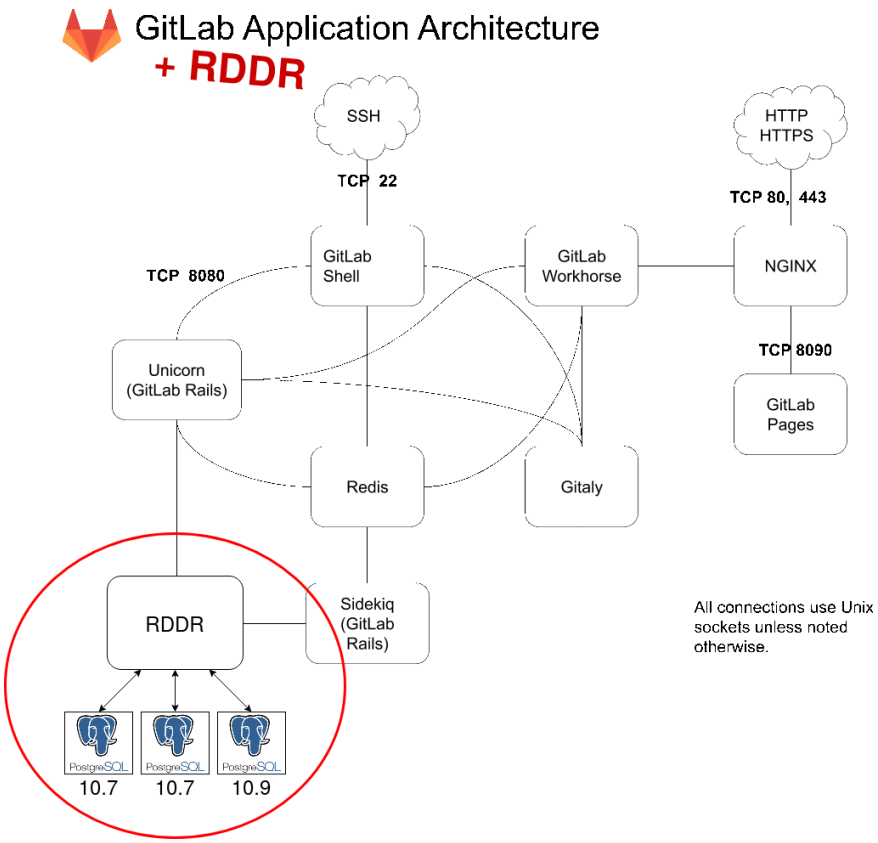


Figure 3.4: Modified GitLab architecture with Postgres replicated behind RDDR.

can then execute the exploit shown in Listing 3.2 to leak the protected rows. First, they create a function which prints its arguments to the console. Then, they create a custom operator to call this function. Finally, they execute a `SELECT` query that invokes their custom operator. The `SELECT` query itself will not return the protected rows, since access control is properly implemented for `SELECT`. However, the function will still be passed the value of column 'a' from every row in the table, and the function can leak them out.

We compose our n-versioned Postgres deployment from three instances of Postgres, two at version 10.7 and a third at version 10.9. Version 10.7 exhibits this vulnerability, whereas 10.9 does not. When the exploit is carried out, RDDR will detect

the difference in behavior between 10.7 and 10.9 and abort the connection. Our modifications to the GitLab architecture appear in Figure 3.4. The two 10.7 instances form the filter pair and enable filtering of noise as described in Section 2.1.2. An empty database is initialized with the schema for GitLab and instantiated in each instance. GitLab is configured to use an external Postgres database and is pointed at RDDR's incoming proxy, which forwards all queries to every Postgres instance. All benign GitLab functions remain fully operational: users can log in, create projects, view projects and more, and RDDR does not interfere. Only when a neighboring container tries to carry out the exploit we described does RDDR jump into action, closing the connection to the client before protected rows from the table are leaked.

This deployment emphasizes that RDDR can function well even in a complex system with plenty of benign traffic. Furthermore, it illustrates how RDDR can be deployed around a specific service or services, rather than n-version the entire application. This enables developers to n-version only the most critical services of their application, thereby maximizing reliability while minimizing overhead.

3.1.6 Performance

TPC-H Benchmark Performance

To quantify the overhead incurred by n-versioning, we will compare the performance of a single instance of a Postgres database to a 3-version deployment (where all three Postgres instances are identical). We evaluate performance using the TPC-H benchmark [62]. The TPC-H benchmark is widely used in industry for comparing the performance of different databases. The benchmark specifies a database schema and 22 queries to test with. We initialized each database instance with a TPC-H database of scale 10x and 10GB of shared memory. We then executed all but one

query against each of our two deployments with 1, 2, 4, 8 and 16 clients in parallel. We skipped query 15 since it cannot be executed in parallel. For each combination of query and concurrent clients, we measured the time to execute, memory usage, and CPU usage of each deployment in order to quantify the overhead incurred for each. We measure the memory and CPU usage of strictly the process tree that makes up each deployment. All testing was done on an AWS virtual machine with 32 vCPUs and 128 GB of memory.

Figure 3.5 shows the overhead incurred by using RDDR. The first plot captures execution time, the second CPU utilization, and the third memory utilization, all normalized to the performance of the baseline single-instance deployment.

We expect the resource overhead of 3-versioning to be approximately 3×. We observe that this largely holds true for memory consumption per the last plot of Figure 3.5. We see a 3× overhead on CPU as well at just one client; the CPU overhead actually reduces with more clients. We attribute this to increased server load as more queries need to be serviced concurrently. This leads to more jobs running in parallel which quickly saturates the available cores for both the single-instance and 3-version deployments.

Figure 3.6 shows the performance breakdown per query for 1 client and for 16 concurrent clients. Note that data for query 15 is missing, since this query was excluded from testing. The complete set of performance plots can be found in Appendix A.

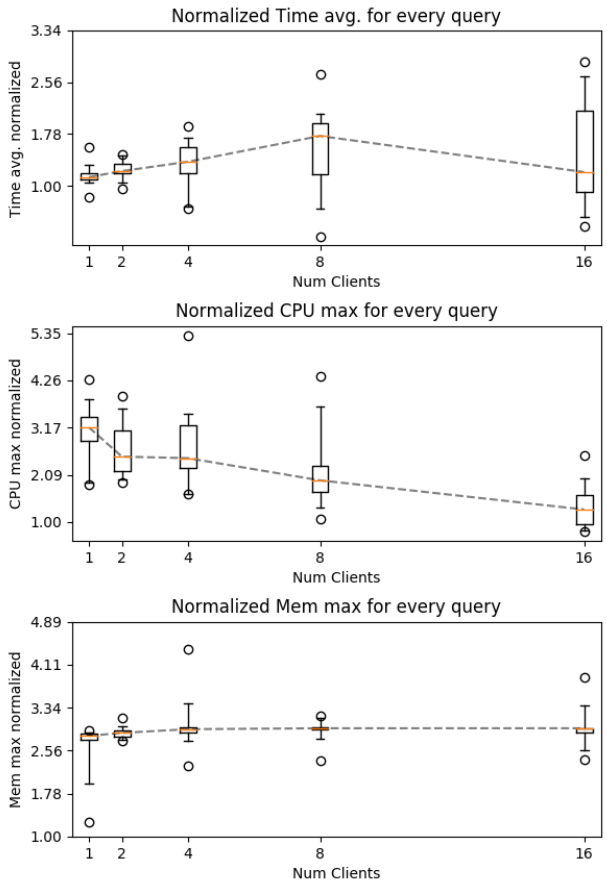
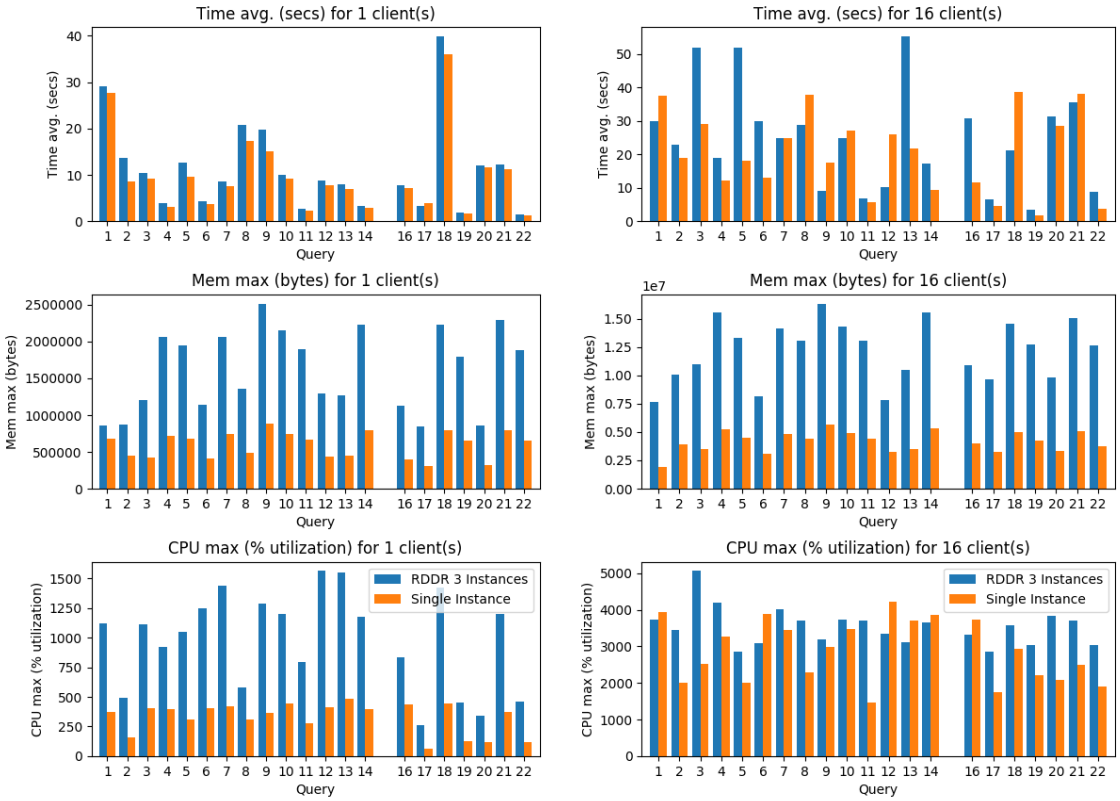


Figure 3.5: Performance of RDDR normalized to the baseline for various different numbers of concurrent clients. Boxes span the 5^{th} through 95^{th} percentile



(a) 1 client

(b) 16 clients

Figure 3.6: RDDR overhead normalized to baseline for 1 and 16 clients.

Throughput and Latency

RDDR is deployed with three identical Postgres instances and evaluated using the `pgbench` benchmark. Its performance is compared to a baseline deployment consisting of a single Postgres instance with and without an Envoy front proxy. We hosted each deployment on an `m5a.8xlarge` AWS machine with 32 virtual CPUs and 128 GB of memory, which we will call the "server machine". The `pgbench` benchmark was executed from a separate `m5a.4xlarge` AWS machine with 16 virtual CPUs and 64 GB of memory, which we will call the "client machine".

First, we baseline the quality of the connection between these two machines. The server and client machines are co-located in the same AWS VPC, and they communicate over the local network such that the number of network hops is one. We evaluate the network connectivity between the two using `netperf`, profiling the throughput and latency of the connection. Table 3.2 collects our measurements. We note a median latency of 1.062 milliseconds and an average throughput of 4726.71 Mbits per second.

Measurement	**Value**
Throughput	4764.27 Mbits/sec
Median latency	0.973 milliseconds
90th pctl latency	1.064 milliseconds
99th pctl latency	1.179 milliseconds
Min latency	0.982 milliseconds
Max latency	2.388 milliseconds

Table 3.2: Baselining network connectivity between server and client machines.

Every deployment was initialized with a database of scale factor 100, or $10,001,100$ table rows across all tables. We run `pgbench` for different numbers of simultaneous clients, ranging from 1 to 256 in powers of two. Each client executes in a separate thread and makes 10,000 SELECT transactions against each deployment. RDDR's

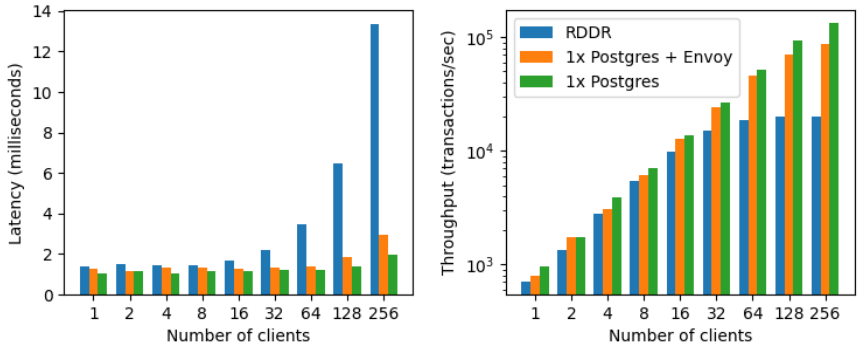


Figure 3.7: Throughput and latency for 10,000 transactions per client.

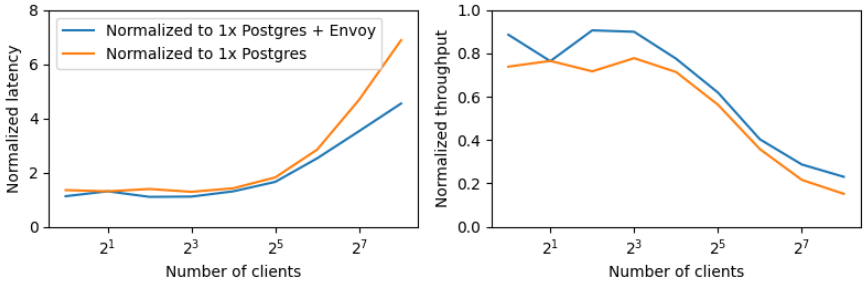
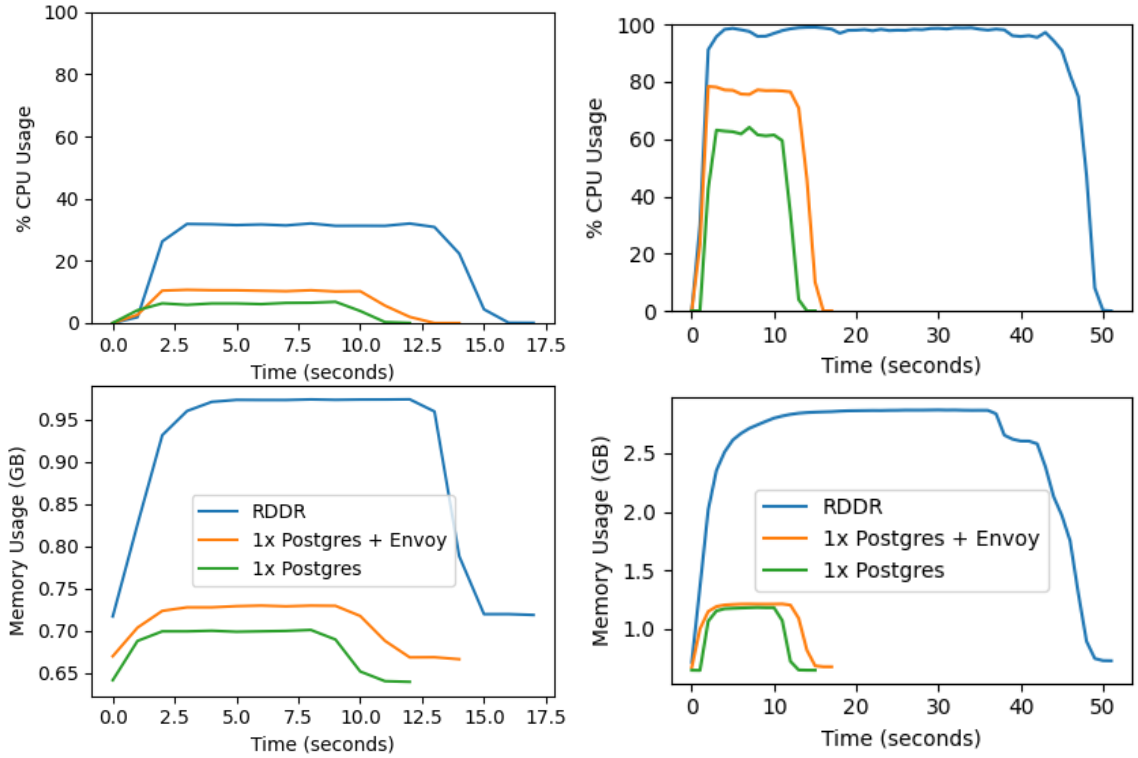


Figure 3.8: Throughput and latency for 10,000 transactions per client, normalized to each baseline.

performance is compared to two different baseline deployments: a single instance of Postgres with an Envoy front proxy, and a single instance without a front proxy. Thus we can understand the penalty RDDR imposes and how that penalty compares to the penalty of adding a front proxy. Figure 3.8 shows RDDR's throughput and latency normalized to each baseline. At 8 clients, RDDR incurs a 10% reduction in throughput and an 11% increase in latency compared to the single instance of Postgres behind Envoy. From there, the host machine begins to become overloaded, and RDDR's relative performance drops.

In Figure 3.9, we quantify the relative CPU and memory usage of each deployment serving 16 and 128 simultaneous clients. We see that at 16 clients, RDDR exhibits roughly 3x memory and compute overhead, as expected for a 3 instance



(a) 16 clients

(b) 128 clients

Figure 3.9: Aggregate CPU and memory usage for each deployment with 16 and 128 clients.

deployment. As we increase to 128 clients, the compute resources of the server become constrained; RDDR is held at near 100% utilization of CPU, while the other deployments are not yet saturating.

We now study the contribution of RDDR's components to this slowdown. Table 3.3 shows the seven different versions of RDDR that we will profile. The first three scenarios, A through C, represent RDDR deployed as a shadow proxy replicating traffic to 1, 2, and 3 different Postgres instances. In the next four scenarios, D through G, we successively enable different features of RDDR on a deployment of three Postgres instances. The throughput and latency were collected for each scenario,

Scenario	# Postgres Instances	Iterates through packets	Compares packet types	Compares packet data	Filters noise
A	1				
B	2				
C	3				
D	3	✓			
E	3	✓	✓		
F	3	✓	✓	✓	
G	3	✓	✓	✓	✓

Table 3.3: RDDR will be profiled in each of the above scenarios to learn the contribution of each component to performance. Only certain features will be enabled in each scenario, indiciated by check mark.

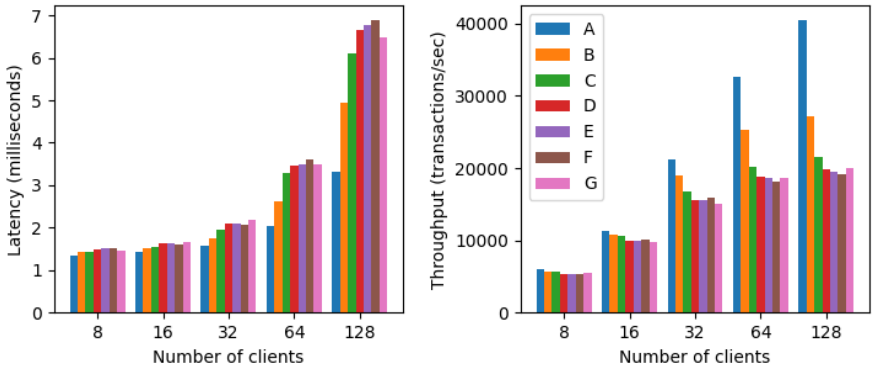


Figure 3.10: Analyzing the performance of RDDR with successive components removed. Each bar represents a different scenario enumerated in Table 3.3.

again executing 10,000 SELECT queries per client against the database. The test was run for 8 through 128 simultaneous clients in powers of two. Results are shown in Figure 3.10.

The measurements exhibit some noise, but the expected trend is visible: as functionality is added to RDDR, penalties to throughput and latency are incurred. Also, as one would expect, the deployments with fewer instances of Postgres scale better beyond 8 simultaneous clients. This is because the CPU overhead to run 1 or 2 instances of Postgres and field 8 simultaneous connection is lower than that

of our typical 3 instance deployment. Scenario A represents the lightest workload for RDDR, yet at 128 clients we still fall short of the single instance with Envoy by 29,000 transactions per second of throughput (-43%) and 3.172 milliseconds of latency (+80%).

3.2 Discussion

In our evaluation we showcased five different example deployments where RDDR demonstrably improves robustness. We also explored the performance of RDDR with two benchmarks. We ran the TPC-H benchmark and found the memory and CPU overhead for one client to be in line with our expectations—about a 3x increase for a 3 instance deployment. With the `pgbench` benchmark, we were able to profile the throughput and latency of RDDR as compared to a single instance of our database with and without a front proxy. This revealed that, on our 32-core server, RDDR's throughput began to lag behind our baselines at 16 simultaneous clients and beyond. This is explained by the additional CPU usage incurred by running three redundant instances of the database engine; simply put, RDDR exhausts the parallelism of the server more quickly than our baselines. Ways to mitigate this issue include upgrading to servers with more cores, or deploying each instance of the n-versioned set on different machines. Though not explored in this paper, it's trivial to spread an n-versioned deployment across multiple hosts with RDDR. Simply alter RDDR's configuration file to point to remote hosts instead of localhost. We also observed that RDDR acting as a simple proxy (Scenario A of our piecewise evaluation) fell short of the performance of the Envoy proxy at a high volume of traffic. This motivates improvements to RDDR's proxying capabilities in the future.

Chapter 4

Conclusion

4.1 Conclusion

In this paper, we presented RDDR, a generic proxy service that enables n-versioning microservices within a cloud application. We argued that n-versioning can help fix a number of common web app vulnerabilities and demonstrated as much via practical deployments. We also measured the overhead inherent in our system and found that we achieved the expected resource consumption for a 3-version deployment, with modest overhead when the parallelism of the host machine has not been exhausted. In the following section, we present potential avenues for future work.

4.2 Future Work

4.2.1 Database Checkpointing

In this paper, we discussed a number of deployments where we n-versioned databases. In these deployments, maintaining the consistency of data between the N database instances can become a problem. If an attacker tries to drop tables from the

database but is only successful for a strict subset of the N variants, RDDR needs to be able to restore the affected variants back to a state consistent with the unaffected variants. We propose in future work to investigate combining RDDR with existing methods of doing checkpoint-and-restore (such as [63–66]) to maintain consistency of data across the N variants.

4.2.2 Alternative Implementations

RDDR could potentially benefit from being implemented as a plugin to existing proxies such as Nginx or Envoy. As we explain in Section 3.1, RDDR's proxying capabilities somewhat lag behind common proxies like Envoy. Were we to leverage Envoy's proxy implementation, and add to it the ability to replicate and compare traffic, it's possible RDDR's performance could be improved. It would also be worth exploring implementations in lower-level languages such as C, Rust, or Go.

4.2.3 Other Variation Mechanisms

Here we propose other ways of introducing diversity into n-versioned web applications that could be explored in future work.

Injecting "Honey Pot" Fields in a Database

While discussing how n-versioning could be applied to OWASP vulnerabilities, we mentioned how "honey pot fields" could be added to database tables so that if an attacker leaks that field, RDDR will detect it. The application will *never* read that field in normal operation, so its appearance signifies an SQL injection. We did not get the opportunity to explore this strategy in this paper, but it would be interesting to evaluate in future work. All of our evaluation thus far has focused on diversity

derived from (i) different versions of the same codebase, (ii) different implementations of the same specification, and (iii) the OS.

Using RDDR to do Permissions Checking

Many of the vulnerabilities affecting GitLab itself are access control related [67–70]. That is, the app simply does incorrect permission checking before sending along some resource. This is subtly different from an information disclosure like leaking stack data, since the data that is leaked is intended for an end-user, just not this particular recipient. Therefore, we can't simply vary the data itself, since in most cases that data will need to be sent out to the client and should not trigger RDDR.

We must find some way to leverage n-versioning to do permissions checking. Here we propose one strategy that is potentially more straightforward than peppering the codebase with permissions checks. Let's suppose the developer is implementing an API which returns a JSON object. He or she can simply add a field to the JSON dictionary which diverges across the instances when the user does not have the necessary privileges to access this data. It could be implemented as an integer calculated like so:

$$f(x) = \frac{|x| + x}{2}$$

$$I = f(p_{required} - p_{user}) \times \mathrm{ID}_{inst}$$

where ID_{inst} is the ID of this instance of the application, p_{user} and $p_{required}$ are the permission level of the user and that required by the system for this feature respectively encoded as integers, and I is the integer which will diverge only when

the user has insufficient permissions. Note that only when the user has insufficient permissions, that is $p_{user} < p_{required}$, is I non-zero. I can be rewritten as follows:

$$I(p_{user}, p_{required}) = \begin{cases} 0 & p_{user} \geq p_{required} \\ \text{ID}_{inst} & \text{otherwise} \end{cases}$$

If the instance ID appears in the JSON returned, the responses diverge and RDDR catches the leak. This technique could be more rigorously evaluated in future work.

Integration Into the Service Mesh Data Plane

In this paper, we targeted a microservice architecture such as the one used by GitLab which does not feature a service mesh like Istio [71]. A service mesh adds infrastructure to the standard microservice deployment, creating more indirection and abstracting the deployment logic from developers' application logic. Part of this infrastructure is the data plane, which consists of proxies deployed as sidecar containers for every microservice in the application. These sidecar proxies abstract away the deployment topology. Each microservice now communicates with other "logical" microservices rather than concrete instances. RDDR could be a useful addition to the service mesh data plane. Microservices can be n-versioned together within a pod with RDDR deployed alongside as a sidecar. This helps to hide the fact that the microservice is n-versioned—an unnecessary detail from the point of view of the other microservices in the deployment.

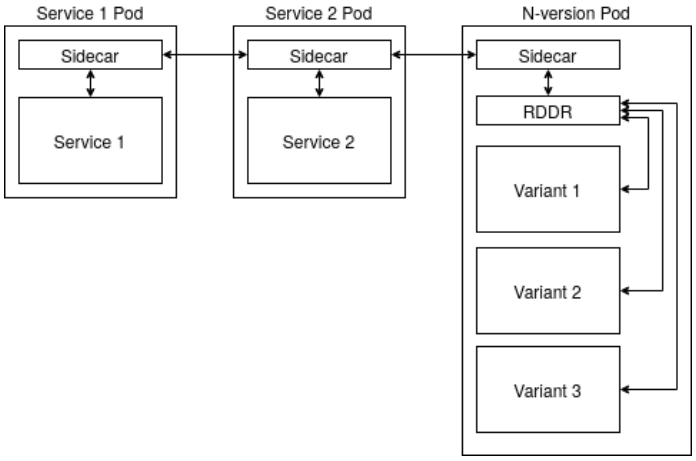


Figure 4.1: RDDR integrated into a service mesh as a sidecar of an n-versioned pod.

Appendix A

Additional Performance Plots

Figures A.1 to A.5 show the performance of RDDR on the TPC-H benchmark compared to a baseline single instance for increasing numbers of clients. Performance is measured in terms of time to execute each query, the maximum system memory in use while executing the query, and the maximum CPU in use while executing the query.

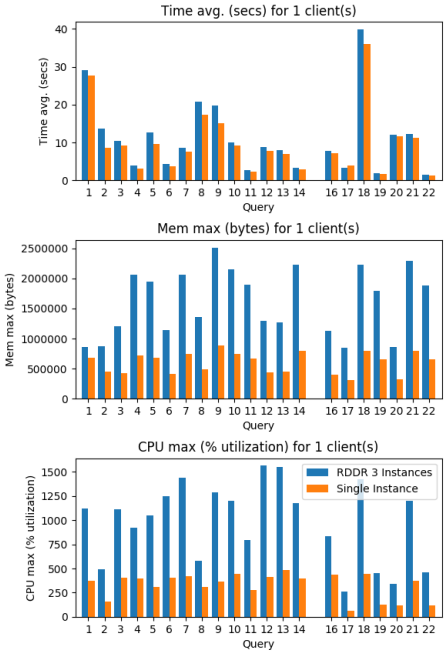


Figure A.1: RDDR performance compared to baseline for 1 client.

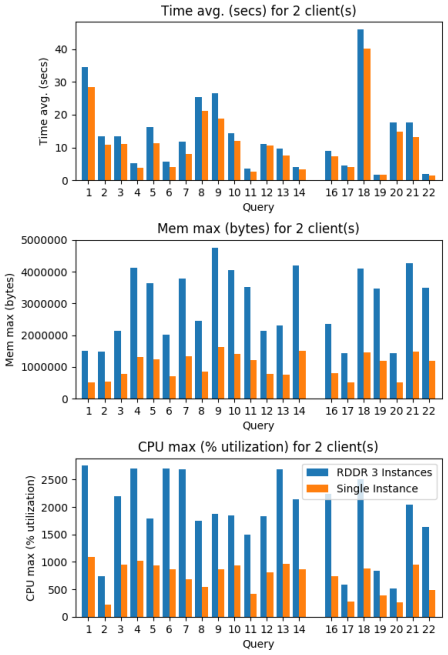


Figure A.2: RDDR performance compared to baseline for 2 clients.

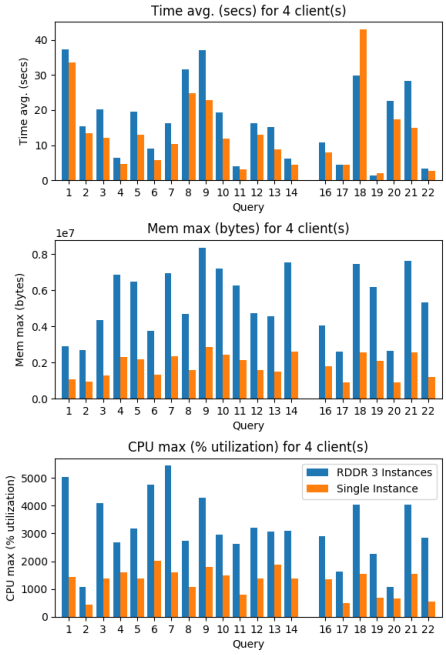


Figure A.3: RDDR performance compared to baseline for 4 clients.

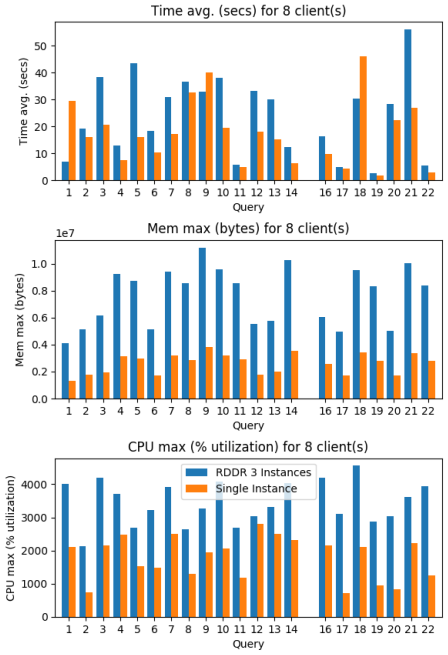


Figure A.4: RDDR performance compared to baseline for 8 clients.

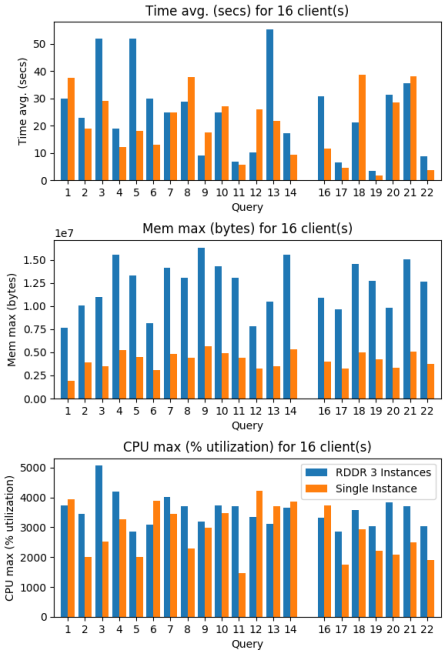


Figure A.5: RDDR performance compared to baseline for 16 clients.

Appendix B

Source Code and Documentation

This project is open source under the MIT License. The project homepage is hosted at:

`https://rddr.readthedocs.io/`

and the source code can be found at:

`https://bitbucket.org/rddr-team/rddr/src/master/`

Bibliography

[1] "Gitlab architecture overview," https://docs.gitlab.com/ee/development/architecture.html#component-diagram.

[2] M. Franz, "E unibus pluram: massive-scale software diversity as a defense mechanism," in *Proceedings of the 2010 New Security Paradigms Workshop*, 2010, pp. 7–16.

[3] D. K. Rensin, *Kubernetes: Scheduling the Future at Cloud Scale*. O'Reilly Media, 2015.

[4] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 26–36. [Online]. Available: https://doi.org/10.1145/2025113.2025121

[5] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel, "Staged deployment in mirage, an integrated software upgrade testing and distribution system," *ACM SIGOPS Operating Systems Review*, vol. 41, 10 2007.

[6] GitLab Inc., "GitLab," https://about.gitlab.com/.

[7] J. B. Hong and D. S. Kim, "Assessing the effectiveness of moving target defenses using security models," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 2, pp. 163–177, 2015.

[8] K. A. Torkura, M. I. Sukmana, A. V. Kayem, F. Cheng, and C. Meinel, "A cyber risk based moving target defense mechanism for microservice architectures," in *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, Dec 2018, p. 932–939.

[9] L. Wang and K. S. Trivedi, "Architecture-based reliability-sensitive criticality measure for fault-tolerance cloud applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 11, pp. 2408–2421, 2019.

[10] N. Smithline, B. Glas, T. Gigler, and A. van der Stock, "OWASP Top 10," 2017. [Online]. Available: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

[11] Y. Xia, Y. Liu, C. Tan, M. Ma, H. Guan, B. Zang, and H. Chen, "Tinman: Eliminating confidential mobile data exposure with security oriented offloading," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2741948.2741977

[12] B. Sullivan, "Xml denial of service attacks and defenses," *MSDN Magazine*, Nov 2009. [Online]. Available: https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/november/xml-denial-of-service-attacks-and-defenses

[13] J. Jones, J. D. Hiser, J. W. Davidson, and S. Forrest, "Defeating Denial-of-Service Attacks in a Self-Managing N-Variant System," in *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-*

Managing Systems (SEAMS). Montreal, QC, Canada: IEEE, May 2019, pp. 126–138. [Online]. Available: https://ieeexplore.ieee.org/document/8787016/

[14] P. Hosek and C. Cadar, "Safe software updates via multi-version execution," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 612–621.

[15] G. Dobson, S. Hall, and I. Sommerville, "A container-based approach to fault tolerance in service-oriented architectures," in *International Conference of Software Engeneering. Citeseer*, 2005.

[16] H. Abdeldjelil, N. Faci, Z. Maamar, and D. Benslimane, "A diversity-based approach for managing faults in web services," in *2012 IEEE 26th International Conference on Advanced Information Networking and Applications*, Mar 2012, p. 81–88.

[17] A. Gorbenko, V. Kharchenko, and A. Romanovsky, "Using inherent service redundancy and diversity to ensure web services dependability," in *Methods, Models and Tools for Fault Tolerance*. Springer, 2009, pp. 324–341.

[18] S. Gholami, A. Goli, C.-P. Bezemer, and H. Khazaei, "A framework for satisfying the performance requirements of containerized software systems through multi-versioning." ACM, Dec 2019.

[19] C. Otterstad and T. Yarygina, "Low-level exploitation mitigation by diverse microservices," in *Service-Oriented and Cloud Computing*, ser. Lecture Notes in Computer Science, F. De Paoli, S. Schulte, and E. Broch Johnsen, Eds. Springer International Publishing, 2017, p. 49–56.

[20] S. Österlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida, "kMVX: Detecting Kernel Information Leaks with Multi-variant Execution," in *ASPLOS*, Apr. 2019. [Online]. Available: Paper=https://download.vusec.net/papers/kmvx_asplos19.pdfWeb=https://www.vusec.net/projects/kmvxCode=https://github.com/vusec/kmvx

[21] R. Huang, H. Zhang, Y. Liu, and S. Zhou, "Relocate: a container based moving target defense approach," in *The 7th International Conference on Computer Engineering and Networks*, vol. 299. SISSA Medialab, 2017, p. 008.

[22] M. Azab, B. Mokhtar, A. S. Abed, and M. Eltoweissy, "Toward smart moving target defense for linux container resiliency," in *2016 IEEE 41st Conference on Local Computer Networks (LCN)*, 2016, pp. 619–622.

[23] Twitter, Inc., Mar 2020. [Online]. Available: https://github.com/twitter/diffy

[24] "Common Vulnerability Scoring System v3.1: Specification Document," FIRST.org, Inc., Tech. Rep. 3.1, June 2019. [Online]. Available: https://www.first.org/cvss/v3.1/specification-document

[25] "OWASP risk rating methodology," https://owasp.org/www-community/OWASP_Risk_Rating_Methodology.

[26] A. Avizienis and L. Chen, "On the implementation of n-version programming for software fault tolerance during program execution," 1977.

[27] J. P. J. Kelly, A. Avižienis, B. T. Ulery, B. J. Swain, R.-T. Lyu, A. Tai, and K.-S. Tso, "Multi-version software development," *IFAC Proceedings Volumes*, vol. 19, no. 11, p. 43–49, 1986.

[28] J. P. J. Kelly, "Specification of fault-tolerant multi-version software: Experimental studies of a design diversity approach." 1983.

[29] P. Kampanakis, H. Perros, and T. Beyene, "Sdn-based solutions for moving target defense network protection," in *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, 2014, pp. 1–6.

[30] Cloud Native Computing Foundation, "Envoy," https://www.envoyproxy.io.

[31] Nginx, Inc., "Nginx," https://www.nginx.com.

[32] "CVE-2017-7529," CVE Details, Jul 2017.

[33] "CVE-2016-6291," CVE Details, Jul 2016.

[34] "CVE-2016-5114," CVE Details, Aug 2016.

[35] "CVE-2017-9788," CVE Details, Jul 2017.

[36] "CVE-2017-15897," CVE Details, Dec 2017.

[37] "CVE-2017-16024," CVE Details, Jun 2018.

[38] "CVE-2019-1010257," CVE Details, Mar 2019.

[39] "CVE-2019-1010306," CVE Details, Jul 2019.

[40] S. J. Moon, V. Sekar, and M. K. Reiter, "Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. Association for Computing Machinery, Oct 2015, p. 1595–1606. [Online]. Available: https://doi.org/10.1145/2810103.2813706

[41] A. Agache, M. Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D. M. Popa, "Firecracker: Lightweight virtualization for serverless applications."

[42] M. Backes, G. Doychev, and B. Köpf, "Preventing side-channel leaks in web traffic: A formal approach," Feb 2013.

[43] "Deploy to swarm," https://docs.docker.com/get-started/swarm-deploy/, Mar 2020.

[44] R. Wood and T. Espinoza, "Rddr source code," https://bitbucket.org/rddr-team/rddr/src/master/.

[45] "Message formats," in *PostgreSQL: Documentation*, 12nd ed. The PostgreSQL Global Development Group, ch. 52.7.

[46] Cockroach Labs, "CockroachDB," https://www.cockroachlabs.com.

[47] EnterpriseDB Corporation, "EnterpriseDB," https://www.enterprisedb.com.

[48] S. Volckaert, B. Coppens, A. Voulimeneas, A. Homescu, P. Larsen, B. D. Sutter, and M. Franz, "Secure and efficient application monitoring and replication," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016, pp. 167–179. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/volckaert

[49] K. Koning, H. Bos, and C. Giuffrida, "Secure and efficient multi-variant execution using hardware-assisted process virtualization," *DSN*, pp. 431–442, 2016.

[50] R. Gawlik, P. Koppe, B. Kollenda, A. Pawlowski, B. Garmany, and T. Holz, "Detile: Fine-grained information leak detection in script engines,"

in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. Berlin, Heidelberg: Springer-Verlag, 2016, p. 322–342. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_16

[51] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 276–291.

[52] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano, "A source-to-source compiler for generating dependable software," in *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, 2001, pp. 33–42.

[53] A. Singh, N. Sinha, and N. Agrawal, "Avatars for pennies: Cheap n-version programming for replication," in *6th Workshop on Hot Topics in System Dependability*, 2010.

[54] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, p. 253–267, Oct. 2003. [Online]. Available: https://doi.org/10.1145/1165389.945470

[55] Dewhurst Security, "Damn Vulnerable Web App," https://www.cockroachlabs.com.

[56] "CVE-2017-7484," CVE Details, Jul 2017.

[57] P. G. Bishop, D. G. Esp, M. Barnes, P. Humphreys, G. Dahll, and J. Lahti, "Pods — a project on diverse software," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 9, pp. 929–940, 1986.

[58] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of CCS 2007*, S. De Capitani di Vimercati and P. Syverson, Eds. ACM Press, Oct. 2007, pp. 552–61.

[59] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *Proceedings of CCS 2008*, P. Syverson and S. Jha, Eds. ACM Press, Oct. 2008, pp. 27–38.

[60] S. Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," 10 2005.

[61] "CVE-2019-10130," CVE Details, Jul 2019.

[62] "TPC Benchmark H," Transaction Processing Performance Council, Tech. Rep. 2.18.0, Dec 2018. [Online]. Available: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf

[63] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing*, vol. 65, no. 3, p. 1302–1326, Sep 2013.

[64] J. L. Gersting, R. L. Nist, D. B. Roberts, and R. L. Van Valkenburg, "A comparison of voting algorithms for n-version programming," in *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, vol. ii, 1991, pp. 253–262 vol.2.

[65] R. A. Lorie, C. Mohan, and M. H. Pirahesh, "Multiple version database concurrency control system," Jan 1994. [Online]. Available: https://patents.google.com/patent/US5280612A/en

[66] S. Fukumoto, N. Kaio, and S. Osaki, "A study of checkpoint generations for a database recovery mechanism," *Computers & Mathematics with Applications*, vol. 24, no. 1, p. 63–70, 1992.

[67] "CVE-2019-16170," CVE Details, Sept 2019.

[68] "CVE-2019-15734," CVE Details, Sept 2019.

[69] "CVE-2019-15733," CVE Details, Sept 2019.

[70] "CVE-2019-11548," CVE Details, Sept 2019.

[71] Google LLC, "Istio," https://www.istio.io.